Re-Targetable Tools and Methodologies for the Efficient Deployment of High-Level Source Code on Coarse-Grained Dynamically Reconfigurable Architectures

Mark I. R. Muir

A thesis submitted for the degree of Doctor of Philosophy.
The University of Edinburgh.
October 2009

Abstract

Reconfigurable computing traditionally consists of a data path machine (such as an FPGA) acting as a co-processor to a conventional microprocessor. This involves partitioning the application such that the data path intensive parts are implemented on the reconfigurable fabric, and the control flow intensive parts are implemented on the microprocessor. Often the two parts have to be written in different languages. New highly parallel data path architectures allow parallelism approaching that of FPGAs, but are able to be reconfigured very rapidly. As a result, it is possible to use these architectures to perform control flow in a manner similar to a microprocessor, and thus a complete program can be described from an unmodified high-level language (in particular C). This overcomes the historical instruction-level parallelism (ILP) wall.

To make full use of the available parallelism, existing microprocessor tool flows are insufficient. Data path machines are typically programmed via HDL tools from the ASIC design world. This expresses algorithms at a lower level than the application algorithms are typically developed in. The work in this thesis builds upon earlier work to allow applications to be described from high-level languages, by employing low-level optimisations in the compiler back-end and working from the assembly, to maximise parallel efficiency. This consists of *scheduling*, where known techniques are used to pack instructions into basic blocks that map well to the reconfigurable core (optimising spatial efficiency); then automatic *pipelining* is applied to dramatically improve the achievable throughput (optimising temporal efficiency). Together these can be thought of as "instruction-level parallelism done right". Speed-ups of more than an order of magnitude were achieved, yielding throughputs of 180–380MPixels/s on typical image signal processing tasks, matching the performance of hard-wired ASICs.

Furthermore, conventional software-based simulation technologies for data path machines are too slow for use in application verification. This thesis demonstrates how a high-speed software emulator can be created for self-controlled dynamically reconfigurable data path machines, using a static serialisation of the data paths in each configuration context. This yields run-time performance several orders of magnitude higher than existing techniques, making it suitable for use in feedback-directed optimisation.

Declaration of originality

I hereby declare that the research recorded in this thesis and the thesis itself was composed and originated entirely by myself in the School of Engineering at The University of Edinburgh.

Mark Muir

Acknowledgements

First, I wish to express my sincere gratitude to my supervisors and colleagues who had the stamina to read through several drafts of this thesis and provide me with excellent feedback.

Further thanks go to my supervisors: Dr Iain Lindsay and Professor Tughrul Arlsan. As primary supervisor, Iain Lindsay went out of his way to help expand my knowledge through countless multi-hour discussions throughout the course of my PhD. One must not underestimate the importance of discussing ideas with someone of great intellect, in order to really test ones understanding, and to thoroughly question every assumption and the validity of any claims and conclusions made. Tughrul Arlsan's wide connections in the academic world allowed me to keep my work focussed at the cutting edge, and to meet other teams internationally and exchange knowledge. He also founded the company Spiral Gateway to commercialise the RICA technology and associated software, which allowed my work to be deployed in a commercial environment and subject to the level of rigour that entails.

I thank Spiral Gateway for employing me to do much of this work, and for giving me the opportunity to put my work into real use, by real people. Working in a commercial environment, and with such a dedicated team—both engineering and management—has given me direct, first-class exposure to the experience of running a company, creating and marketing a product, and communicating with customers. I have been fortunate over my time in the company to be able to see our ideas go from concept to a product nearing final tape out.

My colleagues in the University and in Spiral Gateway have been invaluable in providing a fun and intellectually fast-paced working environment, and for being a continuous source of ideas and topics of conversation.

Lastly, my thanks to the Engineering and Physical Sciences Research Council (EPSRC) for funding this research.

Contents

List of figures

List of tables

Abbreviations

ABI Application binary interface. A convention followed by a compiler to ensure interoperability with other programs on a particular platform. Defines the convention of which registers are reserved for special purposes, how arguments are passed to functions, calling conventions, the format of the stack frame, etc.

ADDCOMP A RICA instruction nemonic representing an addition or comparison operation. These require similar hardware, so were later combined into the same cell type.

ALAP As late as possible.

ALU Arithmetic logic unit.

API Application programming interface. A set of functions providing a high-level interface to certain common or low-level functionality.

ARM Advanced RISC machine. A ubiquitous embedded microprocessor architecture.

ASAP As soon as possible.

ASIC Application-specific integrated circuit. Custom silicon created for a particular task.

ASIP Application-specific instruction set processor. A type of microprocessor where application-specific functionality has been provided through additional instructions. Such instructions are typically very complex, performing high-level functionality.

CFG Control flow graph.

CGRA Coarse-grained reconfigurable array. An umbrella term for particular types of dynamically reconfigurable architectures which operate on the word level (rather than the bit level).

CPU Central processing unit. This term can refer to any type of processor that can perform complex control flow.

DCT Discrete cosine transform. A common phase-agnostic mathematical transform used in image and audio compression.

DFG Data flow graph.

DMA Direct memory access. Hardware that allows memory operations such as block transfers to be performed in the background, without continuous intervention from the CPU.

DRA Dynamically reconfigurable array. An umbrella term for data path architectures that are intended to be reconfigured many times during normal operation.

DSP Digital signal processor. A type of embedded processor with an instruction set optimised for performing common signal processing tasks.

FIR Finite impulse response filter. A type of digital filter with no feedback.

FPGA Field programmable gate array. A ubiquitous data path reconfigurable architecture mostly used in system-on-chip prototyping.

FPOA Field programmable object array. A type of coarse-grained heterogeneous data path reconfigurable architecture.

FU Functional unit. The general term for a hardware block that performs the native operations of a particular architectures. These can be individual gates, ALUs, or even microprocessors.

GALS Globally asynchronous, locally synchronous. A design pattern used in highly multicore architectures, to make it conceptually easier to pass information between the cores.

GCC The GNU compiler collection (formerly the GNU C compiler). An open-source retargetable compiler framework.

GIMPLE GNU variant of SIMPLE—an internal representation used in the GCC compiler, based on SSA.

GPL GNU public license. A license under which open-source software can be released, ensuring the right of 'copy-left'.

GPU Graphics processing unit. Custom silicon which performs highly parallel, high throughput operations used in 3-D graphics. Current GPUs are based on arrays of SIMD processors.

HDL Hardware description language. A computer-parseable language used to express the design of hardware in a scalable, modular fashion.

HPC High-performance computing. A field of computing where problems are solved using clusters of computer nodes which run multiple copies of the same program (each operating on different parts of the problem), connected via a high bandwidth network/interconnect.

ID Short for 'identity'.

IIR Infinite impulse response filter. A type of digital filter where the output is a function of the output delayed by some number of iterations (i.e. feedback).

ILP Instruction-level parallelism. Where multiple instructions in sequence can be executed in parallel. C.f. thread-level parallelism, where parallelism is achieved by having multiple independent instructions streams executing concurrently.

IP Intellectual property.

ISP Image signal processor/processing. A series of algorithms that manipulate digital images, to compensate for artefacts in the sensor and optics. Can also refer to an ASIC implementing this functionality.

JUMP A RICA instruction nemonic allowing the program control flow to be affected by modifying the value of the program counter.

LHS Left-hand side (of an equation or relationship).

LLVM The low-level virtual machine. A powerful new compiler framework / optimising linker which operates around an intermediate representation which describes functionality in terms of the instruction set of a highly generic virtual machine.

MAC Multiply accumulate. A common DSP operation.

MDF Machine description file. A file format used to describe a RICA core, in terms of cell types present, instance counts, locations in the array, timing information, and other properties.

MOV A RICA instruction nemonic representing a 'move'—the transfer of data from one register to another. This concept has no direct physical counterpart in the real hardware, but is used to represent fan-out.

MUL A RICA instruction nemonic representing a multiplication operation.

NRE Non-recurring expenses. Design cost.

NUMA Non-uniform memory architecture. A design pattern in computer architectures where different types of memory are present, in separate address spaces. This improves memory bandwidth, but makes programming more difficult.

OEM Original equipment manufacturer.

OS Operating system. Low-level software running on a particular platform.

PC Program counter. A register that controls which instruction/configuration context is to be executed.

PDC Pipeline depth counter. A hardware concept introduced in this thesis, for hardware-assisted (single-step) pipelining on RICA.

RAM Random-access memory.

RGB Red/green/blue pixel format.

RHS Right-hand side (of an equation or relationship).

RICA Reconfigurable instruction cell array. The dynamically reconfigurable architecture targeted by this thesis.

RISC Reduced instruction set computer. Also known as regular (uniform) instruction set computer, or load/store architecture.

RMEM Read memory. A RICA instruction nemonic representing combinatorial reads from data memory.

RRC Reconfiguration rate controller. A type of instruction cell in RICA which controls the program counter, affecting control flow.

RTL Register transfer level.

SIMD Single instruction, multiple data. A type of computer architecture where multiple ALUs perform the same operation on several data sets at once, following a common control flow.

SOC System-on-chip. A form of ASIC where an entire computer (CPU, memory and peripherals) is integrated into a single die or package.

SRAM Static random-access memory.

SRBUF A RICA instruction nemonic representing reading from a stream buffer.

SSA Single static assignment. A representation of data flow graphs inside a compiler, that makes it easier to analyse.

TLM Transaction-level model. A form of hardware model available in SystemC.

ULIW Ultra long instruction word. A type of VLIW processor.

VLIW Very long instruction word. A type of processor with multiple ALUs.

WMEM Write memory. A RICA instruction nemonic representing combinatorial writes to data memory.

Nomenclature

Abstract netlist A netlist describing only which connections exist, but not how those connections are mapped onto the interconnect. This means the timing information is not accurate.

Active register A register is active in a basic block if it is read from or written to in the assembly instructions belonging to that basic block. After the basic block has been parallelised by a scheduler, an active register may not actually need to be written to if the data that it stores is only used inside that basic block.

Assembly Assembly language (also loosely known as assembler). A low-level but human-readable language which directly corresponds to instructions in the target architecture's instruction set. Assembly provides a thin layer of abstraction above machine language, where the instructions and operands would be coded directly in binary.

Basic block A group of assembly instructions that are always executed in sequence. Basic blocks begin with a unique label, which is used to identify them. Basic blocks either end by simply passing control to the next basic block in sequence, or they end with a jump / branch instruction, which optionally passes control to another named basic block. Basic blocks are the smallest indivisible unit of control flow.

Bitstream Raw binary data that will become the contents of a reconfigurable architecture's program memory, in order for it to execute a given program. This is generated by a tool, from the output of a toolchain, beginning with a human-readable language of some sort.

Coarse-grained Refers to the native width of the interconnect of a reconfigurable architecture. Coarse-grained means that the native width is more than 1-bit, therefore functional units will perform word-level operations.

Compilation unit A C source file plus any other files that it `#includes`, and any that they `#include`, etc. This is the scope in which a compiler typically operates in.

Compiler A software tool that converts source code from a high-level programming language (e.g. C) into a lower-level language to be interpreted by a machine. Typically this is assembly, where the instructions correspond to the instruction set of a particular microprocessor architecture, obeying the rules of an ABI.

Configuration context The data completely describing the configuration of a reconfigurable architecture's functional units and interconnect at a given moment in time, in order for it to form a specific set of data paths. Reconfiguration consists of loading a new configuration context. Also referred to as a wide instruction in some literature.

Control flow graph A graph describing how program control can pass between the basic blocks of a program, during execution. The nodes are basic blocks, and the edges are possible directions of control flow.

Data flow graph A graph describing how the operations in a basic block are connected together into data paths. The nodes are operations (mapping to functional units), and the edges indicate data dependencies between the operations.

Data path A chain of connected operations, where the result of one operation is used as an input to one or more dependent operations in the chain. Operations belong to the same data path if there is at least one unbroken path connecting them (involving any number of other operations in between).

Dead register Registers used in the instructions of a basic block to pass information to other instructions of the same basic block, where the lifetime of this information lies entirely within that basic block.

Dependent operations Two operations are said to be dependent if one cannot begin before the other has produced a result. Dependent operations cannot be executed in parallel, but can be executed combinatorially (if the architecture supports this).

Dormant register A register that is not read from or written to in a basic block or step, but which stores information that is used later in the program.

Emulator A software tool that simulates a specific microprocessor and system, where the behaviour is modelled at a high level, generally leading to faster execution.

Fine-grained Refers to the native width of the interconnect of a reconfigurable architecture. Fine-grained means that the native width is 1-bit, therefore functional units will perform bit-level operations.

Functional unit The elements of a hardware array which operate on data. These can be logic gates, ALUs, or even processor cores.

Immediate A small integer constant that can be coded directly into spare bits in an instruction. Immediates are commonly supported in RISC instruction sets.

Independent operations Two operations are said to be independent if neither depends on the result of the other. Independent operations can safely be executed in parallel.

Input register A register that brings information into a basic block, from earlier in the program. This occurs when the operand of an instruction is a register that has not yet been written to in that basic block.

Instruction A nemonic used in assembly to associate a human-readable name with a bit pattern corresponding to a particular operation in a microprocessor's instruction set. An instruction consists of a name, followed by operands. In a RISC processor, the operands are always registers or immediate constants.

Instruction cell The name given to the functional units of a coarse-grained reconfigurable processor, where the functional units correspond in functionality to instructions common in RISC instruction sets.

Kernel An inner loop in a compute-intensive application, which normally runs for many consecutive iterations. In RICA, this term is also used to refer to a subset of these where the

loop body can fit entirely within a single configuration context. This is the most efficient way to execute, as the configuration context only has to be loaded once during the run time of the loop.

Line buffer Another name for a *stream memory*, used in the context of image signal processing, where local storage is normally for lines of the image near the line currently being processed.

Line memory Another name for *line buffer*.

Live register A register that is read from or written to in the instructions of a basic block, where that information is needed later in the program (outside of that basic block).

Mapper A software tool that determines how paths should be rendered on to the reconfigurable interconnect of a particular architecture, in order to achieve the connections described in an abstract netlist, without conflicts. The output is a routed netlist.

Netlist A file describing the connectivity between functional units in a reconfigurable architecture. The file describes the graph for each configuration context, where the nodes are the functional units (cells), and the edges are the connections. Edges can contain properties that describe the path taken along the interconnect.

Operation A logical operation to transform data in some way. Operations typically have two inputs, and produce one result. Operations are represented by instructions in assembly.

Operation chaining The capability of a given hardware architecture to execute dependent operations in the same clock cycle / iteration. This relies on the ability to execute these operations combinatorially—a physical wire brings the result of one into the input of the other. This is impossible on most architectures, since results normally have to be written to registers, and read back in another cycle / iteration.

Output register A register that is written to in the instructions of a basic block, where the value is not subsequently overwritten (clobbered) by another instruction in the same basic block. The stored information may or may not be needed in later basic blocks. Output registers can therefore be *live* or *dead*, respectively.

Pipeline register A register not originally appearing in the instructions of a basic block, which a scheduler infers during pipelining of a basic block. Pipeline registers delay data between pipeline stages, and are inserted along any connection that spans pipeline stages.

Profile A file containing timing information, execution counts, etc. derived from executing a given program on the target architecture (or in a simulation of it).

Reconfigurable A term given to computing architectures that are not hardwired to perform a single function—i.e. they can change the shape of their data paths in order to change the functionality of the device. This can be done electronically, in-field. The hardware consists of functional units and interconnect (called a *fabric*), on top of which data paths are rendered.

Routed netlist A netlist augmented with path information, showing how each of the connections are physically realised on the reconfigurable interconnect of the target architecture.

Scheduler A software tool that converts the basic blocks of a linear assembly into parallel data paths that are to be rendered onto a reconfigurable architecture. In general, a scheduler extracts parallelism from a sequential stream (of operations).

Simulator A software tool that simulates a specific piece of hardware, where the behaviour is modelled at a relatively low level, e.g. register transfer level in an HDL simulator, or the communication between individual functional units of a reconfigurable architecture simulator.

Step Another name for *configuration context*, used specifically for RICA.

Stream memory Local on-chip random-access memory used as local storage in high-bandwidth streaming applications. Stream memory is normally partitioned into multiple banks to increase the bandwidth.

Target architecture The particular computing architecture that a program is intended to execute on.

Temporary register A register inferred during scheduling/parallelisation to store the value of an edge in the data flow graph that has been split across the boundary between steps. The stored information is internal to the basic block in which it was created.

Chapter 1
Introduction

The choice of platform for many modern digital signal processing tasks in embedded systems is often limited to application-specific integrated circuits (ASICs), since off-the-shelf programmable architectures such as DSPs and microprocessors cannot meet the throughput requirements, whereas reconfigurable hardware such as field-programmable gate arrays (FPGAs) require too much area and power.

For applications that demand an element of reprogrammability, streaming processors (such as those offered by Ambric [1] and SPI [2]) are becoming an increasingly attractive solution, which improve on throughput by providing multiple processing elements/cores with an interconnect structure suited to streaming. However, these processing elements—usually based on regular DSP designs—often equate to significant silicon area.

Alternatively, coarse-grained dynamicaly reconfigurable architectures (DRAs) offer a high degree of parallelism, sufficient to achieve high throughputs [3][4]. Thus fewer cores are required for a given application, leading to a much lower area overhead. Coarse-grained DRAs, such as instruction cell based processors [5][6], provide a high degree of instruction chaining inside the core, by allowing arbitrary connections to be made between the various functional units via a configurable routing network. This allows quite complex data paths to be rendered onto the fabric and executed in a single configuration. This makes these architectures particularly suitable to stream processing, as fewer fetches from program memory are required.

The classes of computing architectures and the languages used to program them are covered in chapter 2. The main observation is that high throughput and area efficiency are common properties of data path machines (due to parallelism), whereas ease of programmability and flexibility are common properties of microprocessors and their derivatives (due to arbitrary control flow). To get the best of both worlds, reconfigurable computing typically involves coupling a data path machine with a microprocessor. Most differences are in the degree of coupling between the two.

Data path machines are programmed using tools from the ASIC design world, since this is what they most resemble; whereas microprocessors are programmed from high-level languages. Therefore, the co-processor approach common in reconfigurable computing generally involves having to partition an application such that the data path intensive parts are implemented on the data path machine, leaving the control flow intensive parts to be implemented on the microprocessor.

Recent innovations in the design of data path machines have allowed them to be in control of their own reconfiguration. Furthermore, they can be reconfigured very rapidly (e.g. millions of times per second), which makes them able to achieve control flow similar to a regular microprocessor. Complete applications can therefore be mapped to a single architecture/device,

using a single language and code base. This significantly reduces design time and improves maintainability.

The reconfigurable instruction cell array (RICA [5]) represents a family of dynamically reconfigurable devices: the technology is scalable—from array sizes of tens of cells, to arrays of thousands of cells (or more). However, there is a trade-off involved: very high-throughput tasks are only possible on large arrays, where there is sufficient parallelism available. However, large arrays have larger configuration sizes, which limits the rate at which they can be reconfigured and the number of configurations that can be stored, which in turn affects the ability to execute arbitrary (general purpose) code, such as for control. Conversely, if the device is to be used more like a general purpose processor but with the occasional need to perform parallel operations of moderate complexity (compared to VLIWs), then its ability to execute large data paths is limited.

Since both scenarios have significant commercial applicability, it is important to be able to address both, and any combination in between. Therefore, the C to RICA mapping tools and simulation tools are required to be generic enough to accommodate the complete range of devices, and beyond (for research).

This thesis proposes and demonstrates algorithms that can be used to create tools that attempt to do this. Two areas of the tool chain are considered in this work:

- A scheduler for extracting parallelism from serial code (resulting from a traditional software compiler), and making sure that hardware constraints are conformed to.
- A high-speed software emulator for the target architecture, designed to be easily extended with new instruction cells and hardware functionality. This is the focus of chapter 3.

Scheduling is further split into the following components:

- Register and resource constrained scheduling of serial code onto parallel architectures supporting operation chaining. Chapter 4. This is how time-division multiplexing is achieved.
- Data path pipelining—exploiting existing hardware techniques to automatically convert a static combinatorial data path into a pipeline, then make use of rapid reconfiguration to apply software pipelining techniques to this pipeline. A simple enhancement to the hardware is also proposed, to make this more efficient for larger cores. Chapter 5.

Pipelining is the main contribution of this work. It allows for dramatic increases in throughput, sufficient to allow the target architecture to compete with hard-wired ASIC implementations. The scheduling work is needed to create the data sets that the pipelining algorithm works on. Emulation is a key component in providing feedback-directed optimisation, for use in improving the scheduling and code layout.

To expedite the development of applications targeting these coarse-grained DRAs, it is desirable to program them from the same languages that the developers use to prototype their algorithms. This is most often high-level languages such as C. The configuration contexts of

such an architecture, if considered to be analogous to instructions in a conventional microprocessor, makes it possible to write a back-end to a conventional compiler (such as GCC) to target these architectures. The types of instruction cell resources in the array are (by design) similar to the instructions available in a conventional RISC (load/store) instruction set. It is these that are represented by instructions in the compiler. Despite being able to reconfigure very rapidly, each configuration context must do a lot more work than a single instruction, as the switching time is still several orders of magnitude slower than that achievable with a conventional microprocessor. Therefore, multiple instructions produced by the compiler must be mapped to each configuration context. This process is called *scheduling*, and is the focus of chapter 4.

Performance is optimised by attempting to match the size of each kernel—inner loops where most of the execution time is spent—to the available resources, allowing them to fit into a single configuration context. This allows the configuration to persist for many clock cycles, operating on new data on each cycle. This increases throughput, since no time is spent having to reconfigure the core between successive iterations. It also decreases power consumption, as the configuration only needs to be fetched from program memory (or cache) once—upon first entering the kernel—rather than on every iteration. However, the resulting data paths can often have a long critical path, leading to poor temporal utilisation of the functional units, since they have to wait until all functional units have completed before operating on the next batch of data, which limits the throughput.

Pipelining provides a way of starting to operate on a new batch of data before an old one has completed. Thus, this allows the functional units of multiple stages of the kernel to be active concurrently; each operating on a different batch of data. Others have devised loop pipelining techniques for reconfigurable architectures [7, 8, 9], where successive iterations of the loop are replicated in hardware, and offset from each other to deal with any data dependencies between the iterations. These are most suitable for large reconfigurable architectures with much longer reconfiguration times, where there are sufficient resources for the entire loop body to be replicated many times. The technique allows complete kernels that were mapped to a single configuration context, to have their critical path length decreased by the addition of pipeline stage registers. Chapter 5 presents two approaches to pipelining dynamically-reconfigurable arrays. In the first, pipeline filling and flushing are achieved through dynamic reconfiguration. This is an entirely software approach. In the second, changes are made to the hardware, to allow filling and flushing to be incorporated into the single kernel configuration context, thus significantly reducing the program memory overhead (especially for very deep pipelines).

Pipelining was demonstrated to achieve significant improvements in throughput—up to an order of magnitude in the examples presented in this thesis. The achievable speed-up scales with the size of the data paths and the size of the core. For use in image signal processing, throughputs of 180–380MPixels/s were demonstrated, which is competitive with hard-wired ASIC solutions. The emulator presented in this thesis is several orders of magnitude faster than competing software-based solutions, and for small cores, approaches real-time performance. This makes it practical for use in feedback-directed optimisation.

1.1 Publications

The work of this thesis is backed by the following publications:

- *"Automated Dynamic Throughput-constrained Structural-level Pipelining in Streaming Applications"*[10]: Presents an algorithm and methodology for the automatic pipelining of configuration contexts on dynamically reconfigurable data path machines. The method utilises dynamic reconfiguration to perform pipeline filling and flushing, to avoid changes to the hardware. The user specifies a timing constraint (i.e. target critical path), and the algorithm constructs as many pipeline stages as needed in order to meet it.
- *"Automatic dynamic structural-level pipelining in reconfigurable processors"*[11]: Improves on [10] by automating the choice of pipeline critical path constraint, allowing for pipelining to be completely automated.
- *"Extensible software emulator for reconfigurable instruction cell based processors"*[12]: Presents a novel serialisation algorithm out of which a data path simulator can be made, and demonstrates its application to reconfigurable computing.

These publications can be found in appendix D.

As of the time of writing, there have been no publications on the scheduling work (chapter 4). This was due to commercial sensitivity of the algorithms employed therein. However, the work provides a foundation for the published work on pipelining. Future publications are planned on some of the novelties described in that chapter.

1.2 Novelty

This thesis presents the following contributions to knowledge, grouped by purpose, the most significant first.

1.2.0.1 Improving throughput of computationally intensive inner loops by pipelining

Dynamic pipelining: (section 5.3) Proposes the idea of rendering custom pipelines on to looping configuration contexts to reduce the effective critical path, thus increasing performance. Dynamic reconfiguration is used to perform pipeline filling and flushing.

Pipeline stage allocation algorithm: (section 5.3.1) Describes the algorithm used to determine where to insert pipeline stage registers into data paths on a coarse-grained reconfigurable architecture, in order to meet a given target critical path constraint.

Single-step pipelining: (section 5.5) Proposes hardware modifications that allow filling and flushing to be performed via the same configuration context as the pipelined loop context, to reduce the memory footprint and increase the number of pipeline stages that can be used. This involves a methodology for describing the side-effects of high-level operations on coarse-grained devices.

Automating the choice of timing constraint: (section 5.6) Shows how the timing constraint can be derived automatically, such that the maximum possible performance can be achieved whilst minimising the resources used.

Support for internally pipelined cells: (section 5.7) Describes modifications to the pipelining algorithm that allow a pipeline to be constructed involving cells that are internally pipelined.[1] This is useful for hiding memory latency, and decreases the pipelined critical path, leading to higher achievable throughputs.

1.2.0.2 Programming dynamically reconfigurable architectures using a single code base from a high-level language

Live register identification: (section 4.7.1) An algorithm for determining which registers contain live information at each stage during execution of a program. This is inferred directly from the assembly instructions produced by a compiler. This information can be used to improve the parallelism, by making more registers available for storing temporary values over the boundaries between configuration contexts. It also frees registers for use in pipelining.

Global register reallocation: (section 4.12) Extends upon the live register identification algorithm to track the flow of unique pieces of information through registers. Demonstrates how this can be used to improve routability, by determining when it is safe to move information into different registers that are closer to where the information is used,[2] without affecting the behaviour of the program.

Register starvation avoidance: (section 4.10) The process of packing data paths together into configuration contexts typically involves using more registers than the compiler originally referenced in the assembly. If insufficient registers are available, scheduling fails. A series of methods are described to work around this, by gradually reducing the parallelism until a valid schedule can be formed.

Scheduling algorithm and associated data model: (section 4.9.2) A scheduling algorithm for packing the data paths of basic blocks generated by a compiler into a sequence of configuration contexts on a reconfigurable architecture. This involves replacing the registers chosen by the compiler with wires where possible, or otherwise renaming registers in order to pack the data paths together in parallel.

Support for memory cascading: (section 4.11.2) Proposes an algorithm for analysing the dependencies between memory access operations within a configuration context, allowing dependent combinatorial memory accesses to be cascaded together. This avoids having to split certain inner loops into multiple configuration contexts, resulting in higher performance.

[1] i.e. the result of an operation appears in a different iteration to when the corresponding input values were sampled.

[2] reducing the amount of interconnect needed.

1.2.0.3 High-speed simulation of dynamically reconfigurable architectures

Load-time serialisation: (section 3.1.3) Simulating dynamically reconfigurable architectures that reconfigure many millions of times per second using conventional RTL simulation is inefficient due to the overhead of serialising the data paths at run-time. The key contribution here is to perform this serialisation before executing the program, so that each configuration context is only processed once. This results in a significant performance gain.

Serialisation algorithm: (section 3.3.2) The algorithm used to serialise the data paths of a coarse-grained architecture, and associated data model for storing this serialisation for later execution by an interpreter.

1.3 Structure

Chapter 2 describes the overall background of this work, discussing the concepts behind reconfigurable computing and the tools that are used.

The next chapters—Emulation (chapter 3), Scheduling (chapter 4), and Pipelining (chapter 5)—cover the main body of the work. Each of these follow the same structure: they begin with a high-level description of the problem that they address, including a list of aims and objectives, along with a summary of the novelties that are covered in that chapter. This summary is followed by a review of the background and literature relevant to that work. The main material follows, describing the concepts and algorithms. These are followed by relevant results and analysis. The chapters close with some final words that explain the importance of what was covered, and how that links to the chapters which follow.

Conclusions are presented in chapter 6, which restates the aims, objectives, and novelties, and discusses the results. It also provides closing remarks.

Chapter 2
Background

This chapter provides an overview of the technologies behind reconfigurable computing [13]—both the hardware itself and the software used to develop applications for them. Section 2.3 describes the foundation of hardware and tools that existed before the work of this thesis, and defines the scope of the problem that the work of this thesis is intended to solve.

Digital computation machines can be classified according to certain properties that they exhibit. Some of these properties are largely related, and in opposition; an example being programmability v.s. throughput[1]. If we plot various families of devices along an axis of programmability/throughput, we see that they form a spectrum, as seen in figure 2.1. We shall also consider cost from the perspective of an original equipment manufacturer (OEM) of consumer electronic products (embedded systems) utilising these technologies as part of their product.

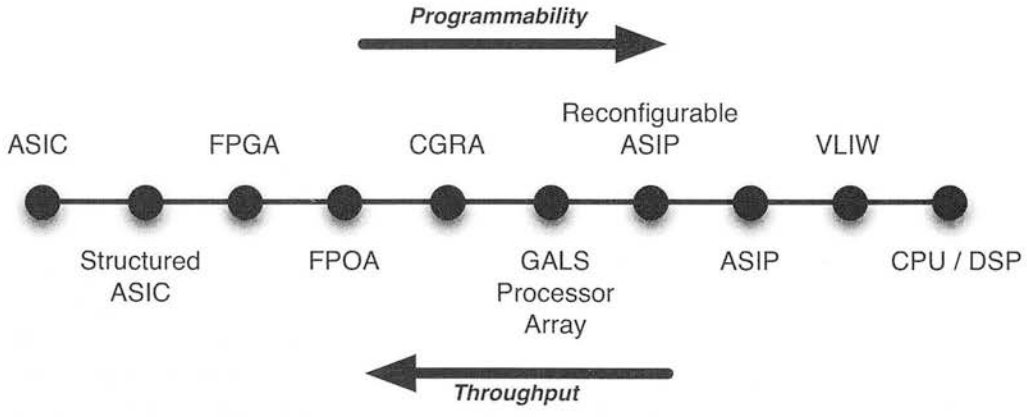


Figure 2.1: Spectrum of devices, from ASIC through to CPU.

[1] also loosely referred to as 'performance'.

2.1 Computing Architectures

2.1.1 Application-Specific Integrated Circuits

At one end sits fully custom devices—ASICs—where the data paths required by a particular application are provided hard-wired in the silicon. ASICs are the pinnacle of throughput[2] and low power consumption[3], but at the same time are the least generic / programmable. Similarly, they have the highest non-recoverable engineering (NRE) costs, and this continues to increase as the silicon processes move to smaller and smaller feature sizes [14]. However, for high volume products, they usually offer the lowest per-unit cost, since the die size will be small, and intellectual property (IP) licensing royalties are likely to be lower since a higher proportion of the design will be in-house.

For relatively low-volume products, the NREs of fully-custom ASIC design can be significantly reduced by using structured ASICs [15]—mask programmable devices. These allow the same parallelism as fully-custom devices, with a little extra area overhead, and reduced performance in terms of power consumption and throughput, due to the standard part (silicon) having more resources than needed and less optimum placements for a given design, and thus additional wire lengths. However, structured ASICs are still fixed-function (i.e. not programmable).

2.1.2 Field Programmable Gate Arrays

Moving further into the spectrum from the left, we come to field-programmable gate arrays (FPGAs) (such as those provided by Altera [16], Xilinx [17], and Actel [18]). These maintain the parallelism available in ASICs, but allow a standard part to be reprogrammed to perform very different functions. FPGAs are fine-grained reconfigurable fabrics, as their interconnect and operations are at the bit-level. This granularity incurs a tremendous area overhead, due to the drivers and switching needed to support the hierarchical interconnect network. This also has an effect on performance—both power and throughput. The NREs are quite low with this approach, as the parts are off-the-shelf. However, as a result of the area overhead compared to ASICs, they come at a high per-unit cost. This is perfectly acceptable for low-volume products, or where re-configurability is a key requirement. In high-volume products where re-configurability is important, but only for a subset of the chip's functions, an increasingly common work-around for the per-unit cost of FPGAs is to use embedded FPGAs as IP blocks in an otherwise custom ASIC.

Another device family that begins to populate the centre of the spectrum is field-programmable object arrays (FPOAs) [19, 20]. These are more coarse-grained data-path machines, which have the potential for reduced area overhead compared to FPGAs, since the coarser granularity reduces the complexity (and area) of the interconnect network. However, they are also less flexible as a result (and are often domain-specific).

The high reconfiguration time for FPGAs and some of the more coarse-grained variants, makes them poor at performing control-flow intensive tasks. Two work-arounds are possible: either statically map all the logic for each path of the control flow into a single configuration, or couple

[2] due to the parallelism of the hardware exactly matching that of the algorithms.

[3] due to direct wire connections and optimum placement.

the device with a microprocessor. The latter involves partitioning the design so that control-flow intensive parts of the application are implemented on the microprocessor, and the data-path intensive parts are mapped on to the reconfigurable fabric. The reconfigurable fabric therefore acts as a co-processor in this arrangement. This approach is becoming increasingly common in high-performance computing (HPC) [21, 22, 23]. However, the co-processor arrangement requires two separate toolchains and design methodologies to be used [24], increasing the development time. This concern has in part lead to the introduction of microprocessor IP blocks (hard macros) in high-end FPGAs, with tool chains that simplify the interactions. The use of hard macros also serves to reduce the package count, and to increase the bandwidth between the processor (hard macro) and co-processor (reconfigurable fabric).

2.1.3 Microprocessors

At the far right of the spectrum lies microprocessors (CPUs) and their variants (DSPs). These represent the ultimate in flexibility, but offer the poorest throughput for arbitrarily complex algorithms. As off-the-shelf components, they offer the lowest NREs, but for high volume products the cost is dominated by the additional area. For use in system-on-chip, most CPUs also incur IP licensing fees.

A lot of work has gone into improving the performance of micrprocessors. One method has been to optimise the layout of the device so as to maximise the operating frequency. This is economically viable for highly general-purpose devices, as the substantial increase in NRE is amortised over a much larger number of sales, keeping the unit cost low. However, there is a fundamental limit to which this is possible, referred to as the power wall [25]. Another method is to increase the amount of available parallelism: the approach used depends on the application domain, and how common particular operations are.

For the most general-purpose CPUs, the approach has been to provide multiple deeply pipelined execution units, each with their own queue (superscalar [26]), and use sophisticated hardware to distribute and re-schedule the incoming instruction stream in order to keep the pipelines as full as possible, using techniques such as branch prediction [27] and speculative/out-of-order execution [28]. These hardware re-scheduling techniques are very costly in area and power [29], and place such processors well out of the price range and power budget for embedded systems.

For less general-purpose, domain-oriented DSPs, parallelism is increased by providing multiple arithmetic logic units (ALUs)—yielding a family of devices called Very Long Instruction Word (VLIW) processors [30] and their derivatives [31]. Compile-time software optimisation techniques are used to make the most of the available ALUs [32]. However, there is a limit to the extent to which instruction-level parallelism can be exploited in general-purpose code (the ILP wall [33]), making these devices specific to particular types of applications, such as those that perform compute-intensive kernels (inner loops), common in digital signal processing. Since such optimisations are static (performed at compile-time), they incur no additional area overhead. As a result, these techniques are more common in DSPs aimed at embedded systems.

2.1.4 Multi-Core

Some high-throughput streaming applications that involve performing the same operations on large sets of data, involve too much state for shared register files (common in VLIWs) to be able to avoid the memory wall [34]. In order to meet the throughput requirements, each ALU must be given a separate (small) local memory, making it almost a fully-fledged processor in its own right. This is Single Instruction Multiple Data (SIMD): arrays of processors, where a single program counter controls an entire group of processors, allowing them to perform the same operation on several different sets of data at once, where each data set maintains its own local state [2]. This type of architecture is particularly well suited to streaming, and is becoming ubiquitous in modern graphics processing units (GPUs) [35, 36, 37].

For larger streaming applications, particularly where there is a lot of control flow, a more complex memory architecture is required, allowing point-to-point communication between the cores. This communication network is often able to feed programs as well as data between the cores, significantly increasing flexibility. Examples include Ambric [1], PicoArray [38], and Tilera [39]. Such processor array architectures tend to use a network-on-chip, or globally asynchronous locally synchronous (GALS) interconnect to improve timing [40] and to simplify the task of programming them (i.e. since these are self-synchronised). From a cost perspective, the area overhead places them well out of the reach of today's embedded systems. However, the level of re-configurability and relative ease of programming makes these very interesting in other roles (e.g. video compression, cellular network equipment). These architectures can also be thought of as a natural extension to ALU-based coarse-grained reconfigurables (described in section 2.1.6), where control logic and local program memories are added to each ALU, making them become full-fledged processors.

2.1.5 Application-Specific Instruction Set Processors

The idea of custom data-paths for direct realisation of a particular algorithm can also be applied to microprocessors. This technology is referred to as application-specific instruction-set processors (ASIPs). The custom data-paths are integrated directly into the instruction execution pipeline, making them appear as special instructions—albeit instructions with a comparatively high latency. These custom instructions can improve the throughput by several orders of magnitude, and significantly reduce the power consumption for these operations.[4] However, since the custom data paths are hard-wired, they cannot be reprogrammed; reprogrammability has to be achieved through use of normal instructions executed in series, which causes a sharp reduction in throughput. Certain application domains have sufficient high-throughput tasks in common that it is commercially viable to provide ASIPs as an off-the-shelf part targeted towards that domain. However, they are more commonly provided as IP along with tools for creating custom instructions for the customer's particular application (such as those provided by Tensilica [41], ARC [42], and Stretch [43]).

A related device family occupies the centre of the spectrum—reconfigurable ASIPs. These replace the hard-wired custom instructions of a regular ASIP with some reconfigurable fabric, onto which custom instructions can be rendered [44]. This makes them similar to an FPGA co-processor (to a master microprocessor), except that the reconfigurable fabric is tightly in-

[4] compared to performing the same task with an equivalent sequence of regular machine instructions.

tegrated into the instruction execution pipeline of the master processor. This partly solves the problem of getting data in to and out of an FPGA co-processor's embedded memory, which is normally the bottleneck in the co-processor approach. The down-side is that the maximum data path size is more limited, which reduces the available parallelism. As a result, to make most efficient use of the available resources, and to justify the additional area incurred by the interconnect, the custom instructions should be modified/replaced at run-time. Some sophisticated run-time scheduling techniques have been demonstrated to achieve this goal [45] by dynamically switching between custom instructions rendered onto the reconfigurable fabric and equivalent sequences of normal instructions, according to the current demand on the system. Reconfigurable ASIPs are also relatively small in area, so are more suitable to embedded systems.

2.1.6 Coarse-Grained Reconfigurable Architectures

Coarse-grained reconfigurable arrays (CGRAs) also occupy the central region of the device spectrum, but approach it from the side of data path machines [46]. They share similar interconnect concepts as FPGAs, but the reduction in granularity results in fewer functional units being needed to implement a given task. The increase in bit width and reduction in functional unit count both lead to a reduction in area overhead.

The functional units of CGRA fabrics tend to be either ALU based [47], or heterogeneous [6]. Heterogeneous arrays can offer reduced area overhead since the silicon utilisation in the functional units per unit time can be higher. This is because the minimum array size for a given design is determined by the operations involved in the data paths of the largest configuration context. An ALU-based array would need at least as many ALUs as there are operations, even though all but one of the operations that an ALU can perform go unused (see figure 2.2). On the other hand, a heterogeneous array could provide largely the same operations as are actually needed. However, the heterogeneous approach complicates resource allocation (i.e. the development tools), and can lead to longer average path lengths.

2.1.7 Dynamically Reconfigurable Arrays

This sets the scene for a newcomer to this centre position—dynamically reconfigurable arrays (DRAs). These are heterogeneous coarse-grained data-path orientated reconfigurable architectures, with built-in control flow capabilities [48].

By moving to coarse-grained reconfigurables, the size of the configuration context decreases, and therefore so does the reconfiguration time (assuming the same bandwidth is available). If the configuration can be made small enough, it becomes practical to save multiple contexts in registers directly in the device. This provides a much higher bandwidth, and allows the reconfiguration time to drop to nanoseconds. With such low reconfiguration times, even control tasks can be performed directly in the reconfigurable fabric. A configuration context no longer has to be a free-standing circuit—the loop body of an algorithm can even be split into a sequence of configuration contexts, each loaded one after another for each iteration.[5]

[5] N.B. only a single batch of data is operated on between each context switch.

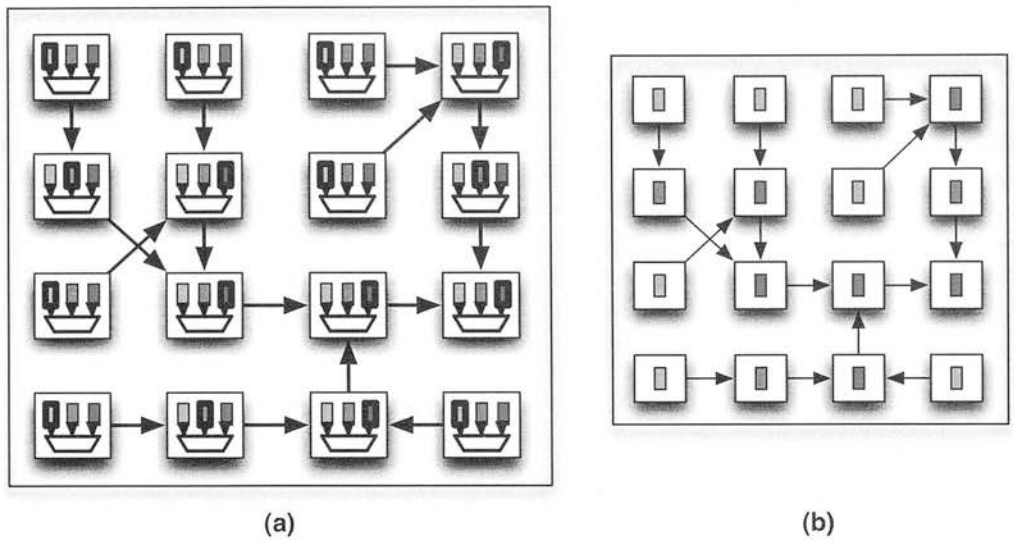


Figure 2.2: ALU-based homogeneous (a) v.s. heterogeneous array (b). Distinct operation types are shown in different colours. The active operations are outlined in (a). To realise the same data-paths, the heterogeneous array is smaller. However, allocation is more constrained in the heterogeneous array, which could result in longer wire lengths (not shown).

2.1.8 RICA

The reconfigurable instruction cell array (RICA) [5, 49], which this thesis focusses on, is a heterogeneous dynamically reconfigurable array. A diagram of a simplified RICA array is shown in figure 2.3.

The functional units (cells) are chosen to match the data width and functionality of RISC instructions in a typical C compiler. RICA uses the concept of distributed registers—a significant fraction of the instruction cells are registers. The function units are connected together via an island-based interconnect network, with an array of switch boxes. The array uses a Harvard memory architecture (to maximise bandwidth)—the program memory and data memory are separate. In many application domains, the array can also have special-purpose stream memories (line buffers) which further increase the on-chip bandwidth. Data can be passed in to and out of the array either via the data memory (memory-mapped I/O), or via special-purpose interface cells; the choice of which depends on the application domain and array size.

The array is in control of its own reconfiguration: a special cell in the core—the `jump` cell—provides access to the program counter and allows the data paths to influence program control flow. Since the structure of the core allows arbitrarily complex data paths to be constructed between the available function units,[6] the combinatorial delay of the critical path in each configuration context may be very different. To account for this, a reconfiguration rate controller (RRC) is used to control the length of time (number of master clock cycles) for which the configuration context persists. This value is stored as part of each configuration context. When the

[6] subject to resource availability and routing considerations.

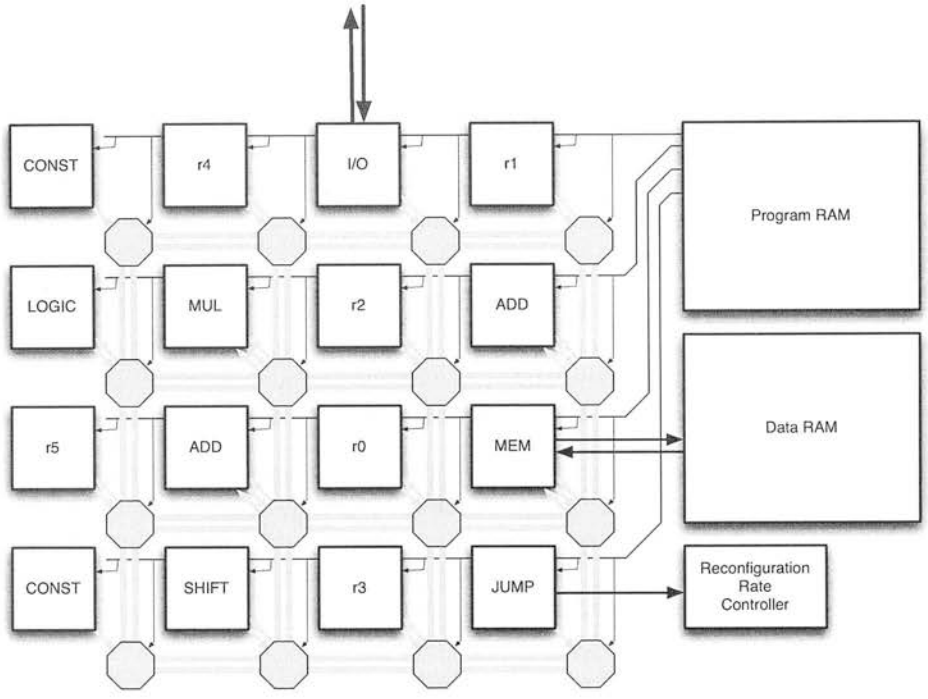


Figure 2.3: Simplified example of a reconfigurable instruction cell array (RICA). The reconfigurable interconnect is shown in grey.

RRC expires, the state of the synchronous cells (such as registers) is updated, and the configuration context referenced by the program counter is loaded. The program counter may refer to the same step as had just ended, in which case that configuration context persists for another iteration, without incurring any transactions from program memory, or any reconfiguration delay. This is the most efficient way to execute loops on RICA; such single-context loops are called *kernels*. If, in a given configuration context, the next value of the program counter can be predicted in advance,[7] the configuration context may be pre-fetched, thus reducing the context switch time. This occurs in configuration contexts that perform an unconditional jump to a constant address.

[7] i.e. if it doesn't depend on values read from the data path in that configuration context.

2.2 Programming Methodologies for Reconfigurable Architectures

Looking at another property—that of design effort for an OEM using these devices—the technologies on the left of the spectrum shown in figure 2.1 are designed/programmed using hardware description languages (HDLs). These offer relatively little abstraction, and expose the application designer to a lot of low-level book-keeping issues such as clock domains and communication timing. As the name implies, these languages are intended for describing data paths and logic—i.e. physically realisable, static circuits.

On the other hand, the various CPU-derived technologies that occupy the right of the spectrum are programmed using high-level languages such as C. This normally equates to significantly lower development times. However, it should be noted that hardware acceleration features such as the custom instructions on ASIPs are designed using HDLs.

Data-path reconfigurable architectures—FPGAs through to coarse-grained arrays—have primarily come from the ASIC world,[8] and as a result have mainly be programmed using tools that work from HDL of some form or another. In general, the more fine-grained an architecture, the larger the configuration size, and thus the longer it takes to reconfigure the device at run-time. With such a limitation, applications have to be mapped either as a static configuration (all-in-one), or as a set of configurations that each must persist for many milliseconds (or seconds) at a time. These long-living configurations have to be free-standing circuits in order to be able to do useful work, and so are still best described by HDLs.

The potential for silicon re-use[9] provided by dynamic reconfiguration has spurred a lot of research into simplifying the task of efficiently mapping a design into a smaller device, using time-division multiplexing of the resources. This includes active research in extending HDLs to support dynamic reconfiguration (JHDL [50]), and software approaches to schedule when a configuration is loaded into a shared FPGA resource, to maximise the useful work that can be done by each configuration. This attempts to work around the problem of the reconfiguration time. However, the large time scale involved in the reconfiguration means that it is still impractical to perform control tasks in the reconfigurable device itself; a master CPU is needed.

The key feature of DRA architectures like RICA is to make use of this by giving the array control of its own reconfiguration, and thus removes the need for a master CPU. This combination of real-time arbitrary control flow and being able to time-division multiplex a loop body makes this architecture suitable to be programmed from a high-level software programming language like C. This essentially merges the two domains of data-path orientated design (HDL) and sequential design (software programming languages). The main drive for this approach is trying to achieve the throughput (and low area) of coarse-grained data-path architectures, with the ease of use of microprocessors.

[8] and marketed as soft ASICs.

[9] and thus the reduction in area for a given design.

2.2.0.1 HDL Based Tool Flows

Tool flows based on hardware description languages consist of an HDL simulator and synthesis (place&route) tools. Examples:

- HDL to gates.
- HDL to configuration bitstream (e.g. for FPGAs).
- JHDL to set of configuration bitstreams (for dynamic reconfiguration of FPGAs).

HDL tool flows suffer from an inherently longer development time than high-level software languages. This is a result of the lower degree of abstraction. Also, due to the search space being large, place&route tools take a long time to run, further leading to long development cycles. Simulation is also slow, due to having to model the entire machine—since the language models the target machine at a low-level, this is the level at which simulation must be performed. This will remain to be the case until tools exist with sufficient intelligence to recognise (and therefore model) more abstract, high-level components, from the low-level description. The primary advantage of HDL is the ability to describe operations that directly conform to a particular time base; high-level software languages have no notion of absolute time.

2.2.0.2 High-Level Software Based Tool Flows

Typical software tool chains consist of a compiler, simulator, and some kind of back-end. Examples:

- C to assembly. The assembly normally matches the underlying instruction set of the target machine. The back-end is a linker.
- C to gates. The back-end normally generates HDL for subsequent place&route.

Use of high-level languages is normally only possible when targeting microprocessors or similar state machines. The flexibility introduced to allow for complex control flow in these architectures results in them having relatively poor throughput. As discussed in section 2.1.3, recent trends in microprocessor designs have lead to ways of increasing throughput by the introduction of pipelines. However, these have the effect of introducing 'momentum'—changes in control flow lead to significant reductions in throughput. This however has also been tackled through branch prediction. However, all of this comes at the cost of substantial silicon area.

C to gates tools (such as [51] or Catapult C [52]) take advantage of the fact that most DSP algorithms have simple control flows, with relatively large areas of computation where there is a significant degree of parallelism. Each node of the control flow graph is mapped in silicon. This approach has the disadvantage of poor silicon re-use, even when partial reconfiguration is used.[10] Therefore, the main overhead of this approach is area. The presence of large data paths and long connections between blocks also sacrifices throughput. Since HDL is the back-end,

[10] due to the high configuration time of fine-grained architectures.

final rounds of testing (from HDL) exhibit the same development cycle as with designs that were HDL from the beginning.

The recent addition of declarative approaches such as Hume [53] have provided another dimension to this, by allowing applications to be designed from a number of different levels: hardware-level (similar to HDL), template/skeleton-level (joining together pre-defined blocks), and high-level (turing complete). This allows the designer to make a different trade-off between ease of description, provability, and efficiency for each part of the design, and to later go back and revise this decision. These are particularly useful in resource-limited or safety-critical systems.

2.2.1 Re-targetable Toolchains

Different CPU architectures exhibit many common properties, which tools can take advantage of: re-targetable toolchains can be created whereby support for a new architecture is added by describing the automaton[11] and to a certain extent stating which properties it exhibits. The tools can then automatically generate an optimising compiler, linker and simulator for the target architecture. Examples of such re-targetable toolchains are LISA [54] and Expression [55].

However, in order to take advantage of operation chaining in reconfigurable computing architectures such as DRAs (section 2.1.7), many of these assumptions no longer apply. If the assembly instructions correspond to operations supported by cells in the core, coarse-grained reconfigurable computing architectures can execute code described in normal assembly in a purely sequential manner just like a microprocessor, with one operation per configuration context. But this would result in very poor core utilisation. Extending this with multiple issue assembly (as used with VLIWs) improves this, but still doesn't take advantage of operation chaining, again leading to poor core utilisation.

2.2.2 Supporting Operation Chaining

In regular assembly, each line represents a change of machine state. With normal microprocessors, each line consists of a single instruction, so the state change corresponds solely to that introduced by that instruction. For use with multiple issue VLIWs, the lines of the assembly can contain certain groups of more than one instruction. The state change is therefore that attributed to all of these acting together [56]. Since the state mostly consists of the contents of the registers, and each line of assembly corresponds to a single change in state of the registers, this means that it is not possible to chain together the operations in a group, since that would require multiple accesses to the register used to join the operations. Furthermore, the latency would be changed. However, some modern VLIWs (or ULIWs [31]) provide limited combinations of operation chaining, which avoid the use of registers, and instead rely on physical connections between the FUs. Again, this is fairly limited, although does increase the possibility of being able to create a single line of assembly that loops back to itself, thus avoiding fetching from memory and decode time.

[11] register classes, supported operations, and pipeline geometry.

To support full and arbitrary operation chaining, we must expand the range over which a state change is expressed: instead of mirroring the state change at each line of the assembly, we should mirror the state change on a larger scale. To maximise the available parallelism, we want to choose as large a range as possible. In order to still meet the control flow demands, the sensible solution is to choose the basic block—i.e. the resulting configuration (or sequence thereof) of the target machine should result in the same sequence of state changes as those corresponding to the end of each basic block in the assembly.[12]

The compiler has to be modified to ensure that new block labels are placed after each jump, so execution cannot leave part-way through a basic block.[13] To help maximise core utilisation, optimisation steps have to be modified to attempt to maximise the size of each basic block [57]. For maximum throughput, each basic block should map to a single configuration context on the target array. However, resource availability and other timing constraints may prevent this from being possible. It is the task of the scheduler to identify these constraints, and to distribute the operations of the basic block across multiple configuration contexts where needed. This will be addressed in chapter 4.

2.2.3 Working From Assembly

With minimal changes to a conventional re-targetable compiler, it is possible to target architectures that support high degrees of operation chaining. This is done by representing the functional units by instructions with identical functionality, and use the concept of the basic block[14] as the basis for constructing one or more configuration contexts.

The assembly for a basic block describes all connections between operations in the data flow graph, using registers as the transport medium. In the ideal case where a basic block maps to a single configuration context, all such internal connections are in fact achieved through the interconnect (i.e. wires), avoiding the register file completely. In order to achieve the correct state of the register file after executing a basic block, it is only necessary to write the final value to each register. All other values written to registers serve only to identify the connectivity, and are turned into wires, or allocated to temporary registers over the boundaries in the resulting configuration contexts. However, there is no direct way of expressing this to the compiler. Furthermore, the compiler has to know how many registers the target supports, in order to determine which edges in the data flow graph may be stored in machine registers, and which on the stack (data memory).

If we specify too few registers, the compiler will make needlessly heavy use of data memory (the stack) to store values. This reduces parallelism,[15] and has a significant effect on latency. It is undesirable to have stack tracking built into the scheduler, as this imposes a need for too much machine-specific knowledge; which reduces re-targetability. If we specify an infinite (or very large) register count, it may be possible for the compiler to generate basic blocks with

[12] i.e. the state after executing the last instruction of the basic block.

[13] to ensure that the operations performed are the same irrespective of the direction of control flow, since the order in the assembly may not be adhered to.

[14] groups of instructions with no control flow in-between.

[15] since dependent memory operations have to occur in more or less the same order as described in the assembly, imposing additional configuration contexts.

too many independent data paths for their initial and final values to be brought in to and out of the basic block via real registers in the core. This would result in the basic block not being physically realisable in the core. The responsibility of deciding which edges in the data flow graph are stored on the stack could be given to the scheduler, but again, this would result in reduced retargetability.

One inefficiency of using the assembly representation therefore is the needless change in value of certain registers that are used only to store temporary values that should really just be wires. However, by analysing the lifetime of each value stored in each register between basic blocks throughout the entire program's control flow graph, it is possible to determine which registers are live, and which just store temporaries. This information can then be used to avoid writing values to dead registers, and thus help reduce power, and free more registers for other uses by the scheduler. In particular, this tends to reduce the chance of register starvation during multi-step scheduling, since final results that are never used do not tie up a register all the way through until the last configuration context generated from the basic block. A method for doing this is described in section 4.7.

The main work of this thesis—creating re-targetable toolchains for rapidly dynamically reconfigurable architectures—chooses to work from assembly. The main reason for this decision is primarily development time: the extensive set of existing language front-ends and middle-end optimisations of GCC can be leveraged, without having to extensively modify the internals of the compiler. Simply writing a back-end is mostly sufficient.

An alternative approach would be to use the compiler's intermediate representation: TreeSSA [58, 59]. However, this was not in a stable state at the time of starting the work on the scheduling. Using TreeSSA may make more information available for scheduling, e.g. memory alias set information, which could reduce the number of configuration contexts produced from basic blocks that involve a lot of data memory activity.

Working from the assembly has the advantage of allowing us to perform scheduling in a separate tool, which:

- Gives more flexibility in adding features (e.g. DFG visualisation, optimised linking).
- Reduces build time (GCC is slow to compile).
- Allows us to change to a different compiler, with little extra work. This allows a range of compilers to be compared, and speeds the adoption of a new compiler if one is found to have more appropriate optimisation passes (e.g. LLVM [60], which emerged after this work began).
- Allows the scheduler to be used as part of a commercial product, without having to release the source code, as demanded by the GPL used by GCC.

However, it also has the following disadvantages:

- Much of the high-level information constructed by the compiler is discarded before reaching the back-end. This makes it difficult to implement optimisations that affect control flow and memory layout.
- The compiler doesn't have direct knowledge of the target architecture, so is unable to make informed decisions about how best to form basic blocks that best match the resources available. This can lead to lower parallel efficiency.

Additionally, pragma compiler directives couldn't be used with GCC, since they apply at the function level, and not at the instruction or basic block level, where the information is needed for the types of optimisations looked at in this work. However, an alternative approach was devised for use with GCC—certain information such as pipelining requests could be encoded as volatile inline assembly hidden behind a human-readable macro. This inline assembly ends up in the same basic block as the operations that it is intended to affect.

Compiler optimisations can be independently developed to improve the generation of basic blocks that are more suitable for use in data path architectures. Future work could integrate the scheduling algorithm and pipelining into the compiler.

2.3 Previous Work

The work in this thesis targets the reconfigurable instruction cell array (RICA), which is introduced in section 2.1.8. The complete toolchain for working with RICA is shown in figure 2.4.

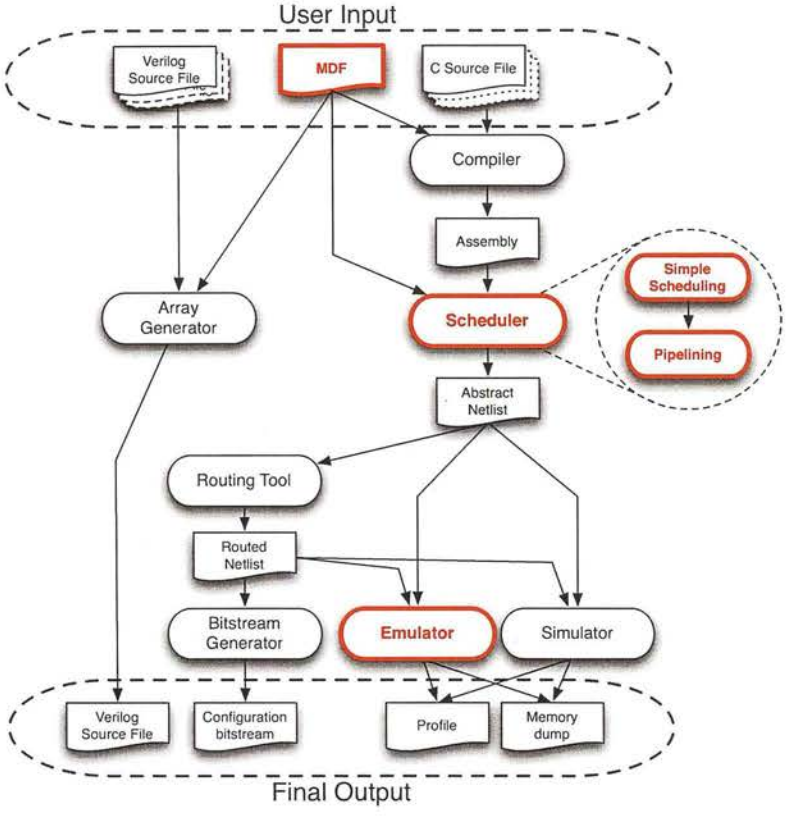


Figure 2.4: Complete RICA toolchain: hardware and software generation. The primary tools and files worked on in this thesis are highlighted in red.

RICA represents a family of related architectures. Prior to the onset of the work described in this thesis, the basic RICA architecture consisted of an array of 32-bit cells, each connected to a switch box (sbox). The switch boxes are connected in a simple 2-D grid, with 32-bit unidirectional interconnect. Each of the four directions has both an input and output channel, connecting it to the neighbour.

A basic set of cell types were provided, which closely match typical RISC instructions (e.g. add, multiply, shift, logic, etc.), along with some special-purpose cells such as data memory access (`rmem`/`wmem`), immediate constants (`const`), and program flow control (`jump`). These cells are often referred to in the text by their corresponding instruction name, expressed in block capitals (e.g. `ADD`). In some variants of the core, certain primitives were combined (e.g. `add` and `comp` → `addcomp`, instruction nemonic `ADDCOMP`). The cells, switch box, and support hardware were implemented as Verilog modules. The array size and individual cell counts are design variables.

```
cell add:
    def ADD
    insts 4
    port out.o
    port in1.i
    port in2.i
    port conf.c.3
    setup 0     // input setup time . combinatorial cells have 0 setup time.
    delay 90    // 0.9ns (expressed in 10ps) 'propagation delay'.

cell mul:
    def MUL
    insts 4
    port out.o
    port in1.i
    port in2.i
    port conf.c.3
    setup 0
    delay 200   // 2.0ns.

cell reg:
    def REG
    insts 32
    port out.o
    port in.i
    port conf.c.3
    setup 15    // 0.15ns.
    delay 25    // 0.25ns.

cell comp_jump:
    def COMP_JUMP
    insts 1
    port n1_out.o
    port addr_in.i
    port comp_in1.i
    port comp_in2.i
    port conf.c.9
    setup 15
    delay 90

channel       // Interconnect.
    setup 0
    delay 80
```

Figure 2.5: Machine description file (MDF) syntax before the work of this thesis. Cell types and their properties and functionality are hard-coded into the tools; just the port layout, timing, instance counts, and arrangement in the core are controlled by the MDF. There is a fixed 1:1 relationship between instruction names and cell type names.

The machine description file (MDF) was created to describe the individual cell counts and what locations they occupy in the array. Timing information and port layouts were also described in the file, for use by the different tools. Each project would typically have its own MDF associated with it. A simplified example is shown in figure 2.5. An *array generator* tool was provided to generate a complete Verilog model of a RICA array, according to a particular MDF.

To generate software to run on the array, a compiler based on GCC with a custom back-end had been created, which allowed C code to be compiled into a RISC-like assembly. A series of scheduling tools were then used to batch the resulting assembly instructions into configuration contexts, to create an *abstract netlist*. This abstract netlist describes just the connectivity between cells, without describing how these connections map to actual paths on the interconnect.

A routing tool [61] exists to render these paths, creating a *routed netlist*. The routed netlist can then be passed to a bistream generator, to generate the configuration bit stream that would be placed in the RICA core's program memory.

This approach of using an existing compiler with a custom back-end, then performing scheduling in a separate tool, was a pragmatic choice designed to get something working as quickly as possible. GCC provides standards-compliant language front-ends, and a wealth of optimisation passes, all ready to use out-of-the-box. The OpenRISC GCC back-end was used as the basis for the RICA back-end, since its instruction set best matched the functionality of RICA's cells. The modifications involved adding support for multiplexers (mapped to C's conditional operator '?:'), and ensuring that labels were placed after each jump instruction, so as to ensure that control flow always starts and ends at the beginning and end of a basic block. The scheduling being performed by a separate tool allowed the flexibility of generating any arbitrary file format (the RICA netlist in this case), and gave freedom to explore ideas without being restricted by the confines of a pre-existing framework. Originally, the scope of the scheduler was very limited, so this approach was the quickest to implement. Later, attempts were made to leverage GCC's own scheduling and register allocation algorithms; however this proved difficult and lead to poor results.

Originally, to test software targeting a RICA array, the entire array would have to be simulated in a conventional HDL simulator tool. This would require the complete software toolchain to be run, and the bistream loaded into the model of the RICA array. Later on, to reduce the design iteration time, a high-level simulation tool (simulator) was provided. This operates on the netlist, and simulates the behaviour of the array, generating execution profiles and a memory dump, for debug purposes. The simulator is a SystemC behavioural model of the target array, which is faster at running than a full RTL simulation of the entire array. Also, the simulator could run on either the abstract netlist or the routed netlist: if accurate timing information is not important, the abstract netlist can be used to test the behaviour of the program, without having to go through the lengthy process of routing each configuration context. This reduced the design iteration time for typical small programs from hours to minutes.

The design space was defined by just the instance counts of each cell type, and their location in the array. Disabling the use of certain cell types, or modifications to the instruction set, required rewriting the compiler back-end and the scheduler. Furthermore, it was only possible to describe combinatorial cells, or synchronous cells with no state. Registers were dealt with as a special case.

2.3.1 The Work of This Thesis

The work described in this thesis expands this design space, by generalising the description of the array. This involved extending the machine description file (MDF) syntax, and passing this information to the various tools. The compiler was modified to read the MDF, to set the register count and disable/enable the use of particular expansion patterns according to the resources available in the target core.

```
architecture
{
    RRC step field width = 10,  // # of bits representing the step persistence time.
    RRC PDC field width  = 10,  // # of bits representing the pipeline depth.
};

architecture cells
{

cell Addcomp (avg)
{
    instruction      ADDCOMP;
    config           conf (width=4);
    input            in1;
    input            in2;
    output           out;
}

cell Mul
{
    instruction      MUL;
    config           conf (width=3);
    input            in1;
    input            in2;
    output           out;
}

cell Mul64
{
    instruction      MUL64;
    instruction      MUL;
    config           conf (width=4);
    input            in1;
    input            in2;
    output           out;
}

cell Reg (register, disjoint)
{
    instruction      REG;
    input            in;
    output           out;
}

cell Jump
{
    instruction      JUMP (must be in first pipeline stage, side effects);
    config           conf (width=4);
    input            addr_in;
    input            cond;
    output           nl_out;
}

cell Sbuf (volatile, disjoint)
{
    instruction      SRBUF (must be in first pipeline stage, side effects);
    instruction      SWBUF (must be in last pipeline stage, side effects);
    instruction      SRBUF_RAM (latency=3);
    instruction      SWBUF_RAM (must be in last pipeline stage, side effects);
    config           conf (width=4);
    input            in1;
    input            in2;
    output           out;
    merge            "'SBUF_SET_READ"    + "'SBUF_SET_WRITE"
                  => "'SBUF_SET_READ_WRITE";
    merge            "'SBUF_STREAM_READ" + "'SBUF_STREAM_WRITE"
                  => "'SBUF_STREAM_READ_WRITE";
}

.
.
.
```

Figure 2.6: Machine description file (MDF) syntax after the work of this thesis. Cell types can be freely described, along with an arbitrary mapping of (multiple) instructions to each cell type, allowing support for cell type aliasing and disjoint operations (i.e. cells that output data independently to their inputs). MDFs can now be cascaded. Only the base ('features') MDF is shown here.

One of the first modifications done to the MDF format was support for cascading—where one MDF can inherit from and specialise another. This was used to partition the information into three layers:

Features: The ports of each cell type, the instructions associated with them, and any special properties that affect their scheduling.

Process: Timing information for the interconnect and cell types, based on a particular manufacturing process node.

Target: Cell instance counts and their layout in the core, for a particular RICA core IP.

This allows common information to be shared between multiple designs, improving the maintainability and readability. Figure 2.6 shows a fragment from the 'features' layer of the new format.

The concept of the scheduler was generalised, so that it could apply to a wider range of devices. This involved devising a data model that was instruction agnostic[16], and adding support for more complex concepts such as the partial aliasing of functionality between different cell types (e.g. `MUL` in figure 2.6, which is supported by `Mul` and `Mul64`), multiple instructions representing the input and output side of a particular cell,[17] (e.g. `SRBUF` for reading from a stream buffer and `SWBUF` for writing to a stream buffer, shown in figure 2.6), or internally pipelined cells (e.g. `SRBUF_RAM` in figure 2.6 which is internally pipelined into 4 stages). Properties were defined for each instruction to define any special constraints that affect their scheduling or how they can fit into pipelines.

Furthermore, to accommodate larger arrays, later designs increased the number of interconnect channels. The MDF syntax was later extended to allow the entire interconnect topology to be described, and this information used by the tools to explore alternative interconnect topologies, for improved routability, performance, and area.

The data model of the scheduler was further utilised to perform more complex modifications such as assembly-level optimisations, and pipelining. The pipelining work has been published on two occasions [10, 11], which are both attached in appendix D.

In addition to this, this thesis describes algorithms that were used to construct a high-speed emulator that could be used as a drop-in replacement for the simulator, so that more complex programs could be tested. The emulator executes target code several orders of magnitude faster, so further reduces the design iteration time from minutes to seconds for simple programs, and makes it feasible to test much more complex programs or larger arrays, and to realistically implement feedback-directed optimisation. This work was published [12], and is attached in appendix D.

The overall effect of this work is that the tools can now address the entire scope of what RICA can be. Additionally, the tools are now fully automated. This minimises the design iteration time, turning these tools into a competitive product for hardware/software co-design.

[16] i.e. could support any type of instruction.

[17] so that these cells can be expressed in the assembly.

2.3.2 Further Work

As the scheduler's role became increasingly sophisticated, the limitations of working from assembly became apparent—certain optimisations need high-level information from the compiler which are lost before reaching the back-end. Examples of this are memory alias sets, loop constructs, and the mapping of variables to registers and the stack. Future work will move a lot of the optimisations presented in this thesis into higher levels of the compiler, where more information is available. This gives further advantages in terms of the ability to re-direct the structure of the code generated according to knowledge of how well the basic blocks will map to the target architecture, and the ability to automatically map variables to different types of memory available in the hardware (such as stream buffers).

2.4 Summary

This chapter presented the background knowledge and related literature to set the scene for the rest of the thesis. Computing architectures were looked at and presented as a spectrum, which reflects the performance v.s. flexibility trade-off inherent in each architecture. A general observation here is that reconfigurable computing—i.e. where a data path machine is used to perform generic computing tasks—generally consists of a data path machine coupled with a microprocessor. This involves partitioning the program, and often requires each part to be written in a different language.

The related software programming methodologies were also presented, showing how languages come from two camps: the ASIC design world (HDLs) and general computing with microprocessors (high-level languages). Recent developments in HDLs have allowed them to express multiple configuration contexts, which helps them express dynamic reconfiguration, which is common in reconfigurable computing. Coming from the other side, there is a lot of recent work in going directly from high-level languages to data path machines (i.e. C-to-gates). This supports rapid application development, whilst sacrificing area efficiency.

Previous work leading up to the work of this thesis was described—the RICA architecture—a data path machine that can control its own reconfiguration, and can be reconfigured rapidly enough to perform arbitrary control flow. This avoids the need for an accompanying microprocessor, and allows an application to be deployed as a single code base in a single (high-level) language.

The next chapters describe the work of this thesis: algorithms and methodologies used to implement a tool chain that allows applications to be efficiently deployed from high-level languages directly to any arbitrary RICA array, and to simulate and profile such applications.

Chapter 3
Emulation

One of the advantages of the reconfigurable cell based processors that are the target architecture for the work described in this thesis, is their ability to be tailored to particular application domains. The potential search space is very large—encompassing physically realisable designs where the metrics of potential throughput and area for a given target application domain can vary by several orders of magnitude.

Simulation is needed to allow for rapid modification and evaluation of the core design, avoiding the time needed to re-implement and test the core using a hardware description language (HDL) for an FPGA implementation, or the cost of re-fabricating the array. Furthermore, these architectures are intended to be provided as flexible IP blocks, where the end-user can make significant changes to the make-up and functionality of the core. The end-user expects a complete toolchain to be available that is able to reflect these changes, in order for the complete hardware/software design space to be explored. Such a toolchain normally consists of an optimising compiler, and a simulator [54, 55]. The application domains that these architectures are mainly aimed at tend to operate on large data sets, such as video playback (H.264 decoding [3]), digital signal base-band processing [4], and image signal processing [10]. As a result, simulation time is a crucial factor in determining the length of the architecture definition cycle, and thus time to market. Finally, a high-speed simulator is necessary to provide feedback-directed optimisation as a standard part of the toolchain.

3.0.0.1 Aims

- Provide a software simulator to allow the design search space to be explored within a reasonable time frame.
- Allow rapid application development and validation.

3.0.0.2 Objectives

Generic: It must be easy to describe the target architecture (e.g. resource counts, timing figures), and the simulation adapt accordingly.

Extensible: It must be easy to add new functionality (e.g. cell types), preferably using a high-level description.

Fast: The simulation should be as close to real-time as possible.

Accurate: The simulation should behave as much as possible like the target architecture at a given level of abstraction, and should give a reasonable estimate of the timing (i.e. within an order of magnitude).

This chapter presents a software emulator that was developed for this class of architecture, satisfying the above goals. Section 3.1 presents an overview of existing emulation/simulation strategies used for reconfigurable architectures and data path machines, and of microprocessors.

Different simulation techniques are for different purposes: a finer granularity of state coherence is needed for early stages of development of a target architecture, to perform more detailed analysis of its behaviour, to determine if and when it deviates from the intended specification. However, once this has been sufficiently tested, this level of detail is no longer required, and instead the focus (and role of simulation) shifts from that of architecture/tool development to target application development. In this latter role, execution speed is of paramount importance.

Instruction-accurate emulators are the fastest form of model available for microprocessors. This is mostly because the state only needs to be modelled as each instruction is executed (i.e. not necessarily cycle accurate). This is only possible because the processors have been designed to make the state be consistent on this granularity, and with sufficient knowledge of processor internals, most other information can be derived from this.

3.0.0.3 Novelty

The novelty of the work presented in this chapter is in providing high speed emulation of a self-controlling reconfigurable data path engine, by making the reconfigurable data paths look like a regular instruction stream—*Load-time serialisation* (section 3.1.3). This is done by first classifying the operations, and then applying a topological sort to perform serialisation—*Serialisation algorithm* (section 3.3.2). This serialisation is performed at load-time—i.e. only once per configuration context—irrespective of how many times and in what order they are executed.

3.1 Background

The task of software simulation of a computer architecture involves the modelling of specific machines on a general-purpose (sequential) computer. Different techniques differ in the granularity of when the state of the simulation matches the state of the target hardware (*state coherence*). This granularity is chosen according to the level of abstraction that is appropriate to what the model is going to be used for. For instance, if the purpose of the model is to debug the operation of the hardware, then a full register transfer level (RTL) simulation is usually necessary. When debugging development tools for the target architecture—which likely require accurate timing in order to validate their output—a full RTL simulation is probably overkill, and a more abstract model that encompasses the timing of the system would be appropriate, such a SystemC model supplied with timing information extracted from RTL simulation. Finally, when debugging applications that are to run on the target architecture, a further level of abstraction is acceptable, where only the functionality and rough timing needs to be captured. This last class of model is called an *emulator*.

3.1.1 Background: Emulation

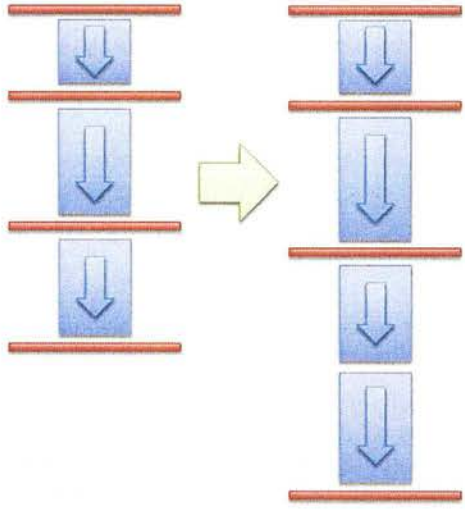


Figure 3.1: Modelling a serial machine on another serial machine: each instruction in the target architecture's instruction set (left) is modelled by an equivalent instruction (or sequence of instructions) in the host architecture's instruction set (right). Blue boxes represent instructions, and the red lines show where the state of the two machines is to match. In emulation, the state must match at the end of each instruction of the target architecture.

Software-based emulation of microprocessors has been used since at least the 1970s [62]. Emulation models the instruction set of the target architecture by mimicking the way that the state of the CPU, registers, and memory is affected by each operation in the instruction set. The fetch and execution of instructions in the emulator is performed in the same sequential manner as in the target CPU. The concept is shown in figure 3.1.

Traditionally, such emulators have been custom-built to a particular target architecture and platform [63]. Since most CPUs are conceptually similar, these concepts can be abstracted, making the emulator extensible. This is commonly achieved through object-orientated design [64, 65]. Emulators are part of many modern commercial tool sets [66]. Emulation sees the following uses:

Behavioural validation: the target architecture and associated application development toolchain can be proven before committing to silicon, or dedicating time to detailed HDL simulation.

Product/Application demonstration: the ability to add emulated hardware allows for applications to be demonstrated in near real-time, before the hardware is available.

Provides an easily modifiable test bench: adding emulated hardware at the behavioural level aids in developing peripherals, since these can be added to the emulator, and their usefulness or interface design explored. This makes it is easy to try out new ideas (platform exploration), without having to design them beyond the behavioural level.

Reduces development time: algorithms can be tested and timing information estimated in a fraction of the time of other software-based simulation techniques available.

Feedback-directed optimisation: information can be extracted about a program through profiling during execution on the emulator. This information can then be used by a compiler [56] to make more informed decisions when applying optimisation [67].

The generalisation of traditional emulation concepts has also extended to the point where emulators can be automatically generated from an abstract machine description, along with an optimising compiler/scheduler as part of a retargetable toolchain [54]. Machine description languages have progressed to the extent that features of increasingly complex architectures can be captured, including deep pipelining of functional units, multiple instruction issue, and the design of the memory subsystem [55]. However, these languages are not yet able to capture the operation chaining available in reconfigurable processors, except by enumerating every possible configuration, which would be impractical. However, such languages could be extended to capture this information, and such a description could be used to automatically generate a simulator using the technology presented in this thesis.

Developments in modern compiler technology have exhausted much of the potential for static optimisation, and so the trend is a shift towards feedback-directed optimisation. As a result, an emulator for this purpose is likely to become a significant part of standard toolchains. With this in mind, the speed of simulation directly affects the scalability of the toolchain with respect to target applications, which are of ever increasing complexity. Hardware acceleration has been commonly explored for use with emulation [68, 69, 70]. However, several of the uses listed above make the requirement of additional hardware undesirable (if not impractical), and so a software-only solution is the main focus of this thesis.

3.1.2 Background: Modelling Data Path Parallelism

The ability of reconfigurable instruction cell based architectures to execute arbitrary control flow makes them similar to microprocessors, if we consider the state changes to be on the instruction level. A configuration context is analogous to an instruction—but one that isn't part of a fixed instruction set, per se. This similarity means that software-based simulation technologies traditionally used with microprocessors can be adapted for reconfigurable instruction cell based architectures, by extending them to take into account the parallelism in the array. However, reconfigurable architectures support operation chaining—the ability to execute dependent and independent instructions within the same clock cycle/configuration context—which traditional emulation technology cannot model.

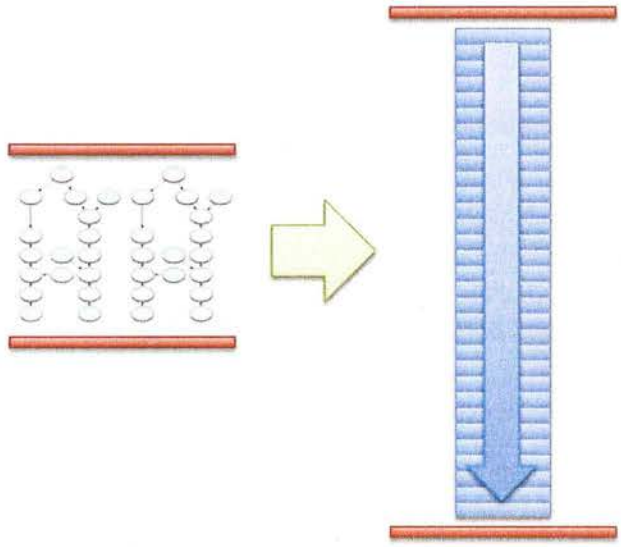


Figure 3.2: Modelling combinatorial data paths on a serial machine: the data paths (left) are broken up into a sequence of instructions (blue boxes) in the host architecture's instruction set (right). The red lines show where the state of the two machines is to match, which is at the end of each complete iteration of the data paths, when their outputs settle. Many of the instructions shown are part of the HDL simulation kernel, which generates the sequence of instructions corresponding to each operation at run-time, according to the events generated.

Modelling parallelism on a serial machine has already been addressed in HDL simulation, particularly those intended for dynamic reconfiguration [50]. The overall concept is shown in figure 3.2. These concepts can be borrowed to derive an event-driven model that captures the data paths between processing elements in the array. SystemC provides an object-orientated event-driven model—called transaction level modelling (TLM)—with a kernel similar to an HDL simulator, but described only at the behavioural level in C.

Such data path machine simulators are slow because they seek to model the target architecture on a very fine granularity (i.e. per operation). The generation and processing of events introduces an overhead each time an operation is to be executed. In the example of RICA, the operations are relatively simple, and often map to only a few host instructions—often much

less than the overhead incurred to generate or process each event. Furthermore, events may be triggered more than once for each operation (due to flutter) before the core stabilises. This gets increasingly worse as the complexity of the data paths increase.

This kernel-based approach of serialising in response to run-time events also imposes an overhead per configuration context. For traditional reconfigurable and dynamically reconfigurable hardware, the rate of reconfiguration is low, so the overhead of updating the event-driven model on each configuration context represents only a small fraction of the total execution time. However, reconfigurable instruction cell based processors are reconfigured many *millions of times per second*, so this overhead introduced by the model is large compared to the actual work done by the operations of the modelled cells.

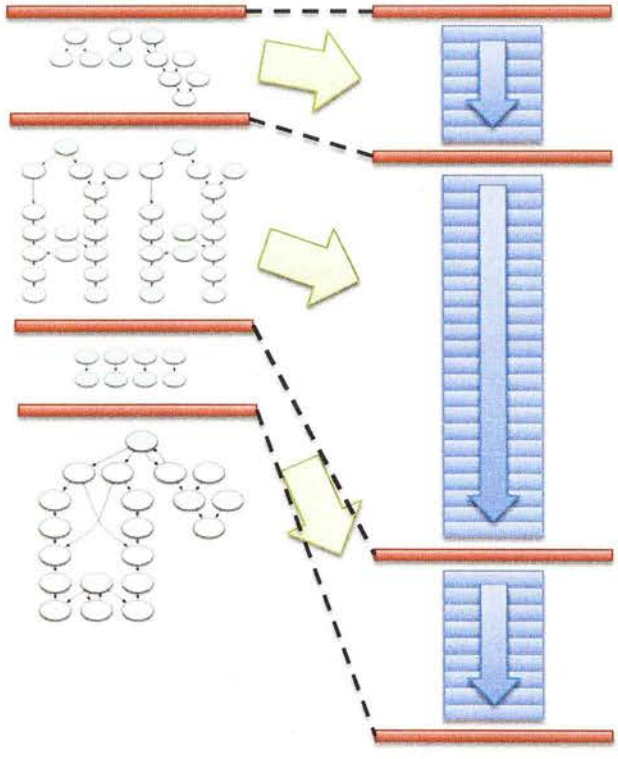


Figure 3.3: Modelling sequences of parallel data paths on a serial machine: each set of parallel data paths (left) is converted at load-time into a series of instructions (blue boxes) in the host architecture's instruction set (right). The red lines show where the state of the two machines must match, and these correspond to basic block boundaries. Within a basic block on the target architecture, the same series of instructions on the host must work with any given data, each time the basic block is called.

Therefore, moving this overhead into a pass prior to program execution is highly desirable, and is analogous to partial evaluation [71]. A program ($prog$) can be thought of as a transform to convert static input data (I_{static}) and dynamic input data ($I_{dynamic}$) into output data (O), i.e.:

$$prog : I_{static} \times I_{dynamic} \rightarrow O$$

Partial evaluation computes a *residual program* ($prog^*$) from the original program and the static input data, that converts just the dynamic input data into the output data, i.e.:

$$prog^* : I_{dynamic} \rightarrow O$$

This is what the software-based emulator presented in this thesis does—it computes the residual program, and executes that at run-time. The emulator moves away from the event-driven approach, and instead mimics the same order of data flow by generating a static schedule of operations that are performed sequentially, as shown in figure 3.3. The algorithm for generating this serialisation, along with the required storage queues, is described in section 3.3.2. This is a new extension to traditional software-based emulator technology, allowing this type of model to work with these unusual architectures.

3.1.3 Contribution: Load-Time Serialisation

The key idea proposed in this chapter is to improve simulation speed by pushing as much work into the pre-execution phase as possible. If we assume that, like with microprocessor emulation, the architecture behaves according to specification, then the required granularity of state coherence is that equivalent to an *instruction*. An instruction in a data path machine such as RICA, can be seen to be the state change resulting from the execution of a single iteration of a single configuration context. The instruction therefore consists of a number of dependent and independent operations executing in parallel, or combinatorially (since operation chaining is allowed). Note however that the term *instruction* is more commonly used to describe the individual operations in the data path (i.e. the functionality of the cells), since these are what are captured by the instructions in the assembly.

If the target program (netlist) is properly formed, the steps will all have been programmed to be given enough time for the final results to stabilise before switching to the next iteration/step, thus making the results deterministic. This is the same as saying that the architecture performs to specification. Therefore, if just the intended functionality of the program is to be simulated, we can assume that the program is properly formed, in which case only the final results of the data paths are needed - i.e. the emulation need only ensure that the state matches that of the real hardware at the end of each step, and not necessarily anywhere in between. These deterministic final results can, by definition, be evaluated by the same sequence of operations each time (given the same initial machine state). This is complicated by the presence of operation chaining, as the operations in each chain must be executed in the correct relative order.

Note that simply recording the sequence of events generated by a simulator is one possible solution to this, but not an ideal one since many operations are executed more than once, which is redundant from the point of view of final state (at the end of the step). Furthermore, there may be a certain extent of data set sensitivity, where some transitions may be missed during recording due to the value not changing for the particular data set used, but where in general, the value could change.

A more efficient approach would be to determine what order to execute each operation in order to achieve the same results. Furthermore, it would be good to minimise the number of operations executed—avoiding redundant executions. The serialisation algorithm proposed in section 3.3.2 consists of a topological sort of 'actions' corresponding to each operation. There are additional 'actions' used to capture synchronous effects such as the updating of registers at the end of a step, and actions to generate initial values.

3.2 The Modelled System

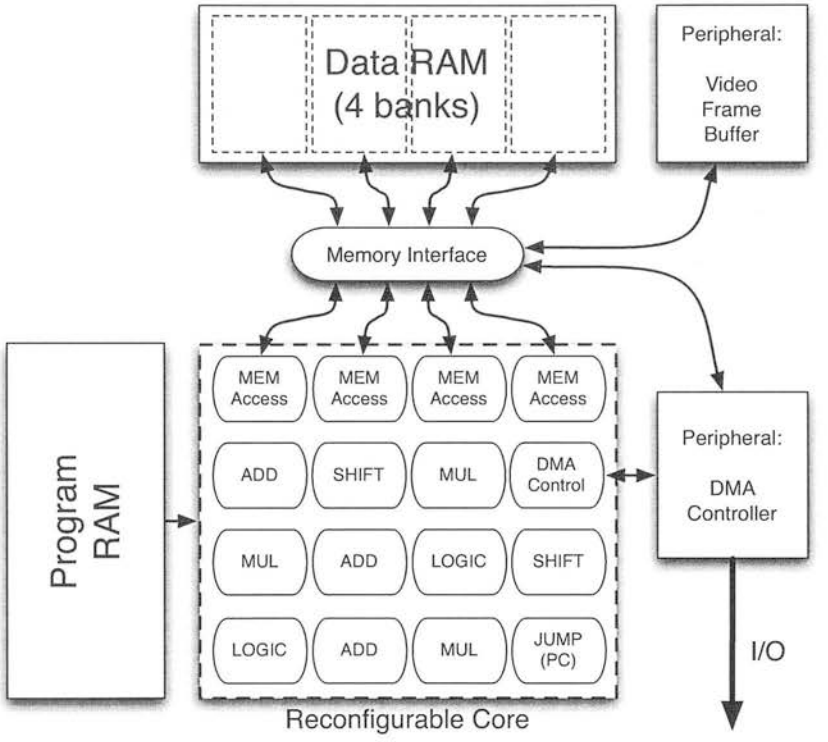


Figure 3.4: Modelled system: reconfigurable core (simplified), memory, and example peripherals.

An example system that can be modelled with the emulator is shown in figure 3.4, and consists of the instruction cell array core, with separate program and data memories, and some simple peripherals. In this example, data memory is arranged in multiple banks, accessed through special cells in the array. Since more than one memory access cell is provided in the array, multiple accesses can be performed by the core in one configuration context. If all such accesses are to different banks, then these accesses are performed in parallel. Otherwise, conflicting requests are performed sequentially, which incurs a dynamic delay.

The emulator can be used to characterise memory access patterns and use the results to direct scheduling and linking of the program (chapter 4) to optimise access to data memory (feedback-derived optimisation, mentioned in section 3.1).

Software emulations of memory-mapped peripherals such as a DMA controller, video frame buffer, or audio buffer can easily be added. These communicate with the core just like they would in real life: either through the memory interface, or through special-purpose cells in the array. New instruction cells can be added to the core simply by defining a new object. New memory-mapped peripheral modules can be added by defining a new object for the peripheral, which responds to events from the memory interface through known method calls, see figure 3.5, in response to activity on the appropriate addresses.

Peripherals can have a kernel that operates on a separate thread, if they are to perform operations that are independent to the core. The video frame buffer emulation is an example of this: it performs colour space conversions and renders frames largely on its own time-base. More complex peripherals, such as a DMA controller, can be added that connect to both the memory interface and to the array via special control cells. These could be implemented by creating a new object for the special cell, and allowing the cell object to communicate with the memory interface. Other scenarios are also possible.

```
object Memory_interface
{
  properties:
  - references to the data memory banks

  methods:
  - readFromAddress:
  - writeData:ToAddress:
}
```

Figure 3.5: Pseudo-code for memory interface.

3.3 Emulator Technology

This section describes the technologies and abstractions involved in the RICA emulator, which give the ability of high execution speed (by reducing the per-step overhead), and allows the functionality to be extended with little effort.

The emulator is an object-orientated program written in C++, and is modular in design. Each hardware component mentioned in section 3.2 is represented by a class (object), and they communicate with each other via method calls. The model of the core is simply a set of instruction cell models, each of which contains the state information that the real cell would maintain, and a set of cell *actions* which capture the behaviour of that cell. The cell actions are implemented as C++ *methods* (member functions). The operation of a given cell is represented by one or more of the following cell actions:

Evaluate: Assign the output value of the cell and/or modify the internal state of the cell according to the configuration word.

Operate: Assign the output value of the cell according to the configuration word and the values read from its input(s).

Update: Modify the internal state of the cell according to the configuration word and values read from the input(s).

A serialised configuration context consists of the *evaluate* actions (scheduled in any order), followed by the *operate* actions (specifically ordered by the serialisation algorithm described in section 3.3.2), followed by the *update* actions (in any order). Cells that perform only simple combinatorial operations—which calculate an output value based on the values of their inputs—implement only the *operate* action. The code sample in figure 3.6 demonstrates a simplified version of an `add` cell, which is an example of a combinatorial cell.

The emulator parses the netlist describing the target program, then serialises the operations of each configuration context into a sequence of equivalent cell actions. These serialised operations are stored in an internal data model. The serialisation process is described in section 3.3.2. Execution of the program then proceeds: these sequences of cell actions for each configuration context encountered are executed in a large state machine by calling the appropriate virtual function, as shown in figure 3.7.

The model of each cell contains a variable that holds the value for the cell's output port. This can then be referenced (read) by the actions of cells that depend on that value (the `input` vector passed into the `operate()` method). Note that the program counter can also be updated via the cell actions (for the `jump` cell), and this determines which configuration context will follow. A configuration context is the smallest unit that can be used as the target for jumps. The data memory is modelled as a simple array wrapped by an object that provides an interface to read and write to the memory, as described in section 3.2.

```
object Add_cell extends Instruction_cell
{
  properties:
  - output // Storage for cell's output.

  constructor:
  - define cell configuration and input ports.

  methods:
  - operateWithConfiguration:inputs:
  {
    switch configuration
    {
      case ADD_ADD_SI: // Single integer.
        output = in1 + in2
      case ADD_SUB_V2HI: // Vector mode.
        output = (in1[1] - in2[1],
                  in1[0] - in2[0])
      etc.
    }
  }
}
```

Figure 3.6: Simplified `add` cell class implementation pseudo-code.

```
// Execute steps until end condition is detected.
do
{
  step index = jumpcell program counter value
  this step = program[step index]
  for each cell action in this step
  {
    switch cell action type
    {
      case Evaluate:
        action.instruction_cell->
                  evaluate(action.configuration)
      case Operate:
        action.instruction_cell->
                  operate(action.configuration,
                          action.inputs)
      case Update:
        action.instruction_cell->
                  update(action.configuration)
    }
  }
}
while jump cell hasn't detected end
```

Figure 3.7: Core execution loop pseudo-code.

3.3.1 Extensibility

To tackle the goal of easy extensibility—where the least amount of effort is needed to later add new functionality to the model—a combination of techniques were used: subclassing, preprocessor meta-programming, templates, and scripting. These will become apparent in the description that follows.

Certain key hardware concepts such as instruction cells and the memory interface were generalised by decomposing them into simple interface descriptions, which were then described

using C++ abstract classes. For instance, the concept of an instruction cell is represented by the abstract class `Instruction_cell`, from which all concrete cell types are derived. A minimum set of concrete subclasses of these were created to describe the particular architecture described in section 3.2. This set of concrete subclasses is intended to be added to by an end-user, to model different systems.

To minimise the cost of design-space exploration, the emulator shouldn't have to be recompiled in order to model a different core. Instead, the core should be defined by a user-provided machine description file (MDF), which lists the types and instance counts of the instruction cells in the core, and other information about the modelled system. To allow the geometry of the modelled core to be defined at load-time, according to the given MDF, the ability to spawn new instances of a particular class at load-time is needed. At load time, the MDF is parsed, and each cell instance defined there is instantiated in the core by requesting a new instance of the specified cell type name.

The concept of a class factory was used to allow a new instance of a cell to be obtained by name. This is achieved by splitting the concept of a cell into two parts: the cell instance, and a corresponding factory. The factory is a separate class, which spawns new instances of the corresponding cell type, and can be queried for other information about the corresponding cell type. An instance of this factory is created for each cell type when the emulator loads. Each cell factory derives from the `Instruction_cell_factory` base class. The base class is implemented as a singleton: at most one instance is allowed to exist. This singleton instance maintains a registry of cell types—or more specifically, records a pointer to the cell factory instance corresponding to each cell type name. Code in the emulator's loading logic deals only with the singleton cell factory instance, requesting from it instances of cells by name.

Each cell type requires a corresponding factory class, in order for that factory to register the cell type with the runtime. As part of the constructor, these factories register themselves with the base class `Instruction_cell_factory`. Since the functionality of each cell factory is identical,[1] these factories are described by a C++ class template, with the template parameter being the cell class to return.[2] This means that to register a cell type with the system, only a single line of code is needed: a file-static instantiation of the factory template, with the particular cell type given as the template parameter.

This process is further automated by a build script, which scans a predefined directory for C++ header files representing cell types, and generates a single header file (`cell-definitions.hh`) that `#includes` each of these.

Certain auxiliary information is needed for each cell type, e.g. to allow user-configurable options to be passed to the cells via command-line options to the emulator, and to define which configurations are supported.[3] This information is referred to in various different places in the emulator's main source code. To maintain readability (and thus maintainability), all of this information is contained in the header file for each cell type. Normally, such information would be provided as part of the cell class implementation, via the C++ virtual function mechanism. However, much of this information is referred to in the main execution loop, which

[1] with just the returned cell class type being different.

[2] i.e. the name of the corresponding concrete subclass of `Instruction_cell`.

[3] and their names, so that these may be substituted in debug information, to improve readability.

is extremely performance critical. Therefore, querying information via virtual function calls in such situations could reduce performance by an order of magnitude or more, considering that the time taken to perform a virtual function call (or even a normal function call) is often much larger than the time taken to execute a cell action. To avoid this, the information is provided via case statements and look-up tables, which are resolved at compile time. To construct these tables and case statements, the C pre-processor is used—a technique called pre-processor meta-programming [72].

```
#if defined INSTRUCTION_CELL

// Define the class name for this type of instruction cell:
INSTRUCTION_CELL(Add_cell)

#elif defined CONFIGURATION_NAMES

// List the supported configuration names for this cell.
CONFIGURATION_NAMES(Add_cell,
                    CONFIGURATION_NAME( ADD_DISABLE )
                    CONFIGURATION_NAME( ADD_ADD_SI  )
                    CONFIGURATION_NAME( ADD_SUB_SI  )
                    CONFIGURATION_NAME( ADD_ADD_HI  )
                    CONFIGURATION_NAME( ADD_SUB_HI  )
                    CONFIGURATION_NAME( ADD_ADD_QI  )
                    CONFIGURATION_NAME( ADD_SUB_QI  )
                    CONFIGURATION_NAME( ADD_ADD_DI  )
                    CONFIGURATION_NAME( ADD_SUB_DI  )
                   )

#elif defined CELL_OPTIONS

// List the options available for this cell.
CELL_OPTIONS(Add_cell,
             CELL_OPTION(di_mode_width, "64",
                         "Sets the bit-width for double integer mode.")
            )

#else

// Configuration name macros, defining the corresponding values:
#define ADD_DISABLE  0
#define ADD_ADD_SI   1
#define ADD_SUB_SI   2
#define ADD_ADD_HI   3
#define ADD_SUB_HI   4
#define ADD_ADD_QI   5
#define ADD_SUB_QI   6
#define ADD_ADD_DI   7
#define ADD_SUB_DI   8

// Cell class definition:
class Add_cell : public Instruction_cell
{
    .
    .
    .
}

#endif
```

Figure 3.8: Pre-processor guarded sections in a typical cell type implementation header file (with class implementation details omitted).

The C++ header file for a cell type is effectively split into sections. An example is shown in figure 3.8. The header file is parsed several times during compilation: once for each section. Each section is contained in a `#ifdef <section-name>` block, where the appropriate section name macro is used. The last section is unnamed (i.e. uses `#else`), and contains the cell type class definition. This is what will be seen when including the header file in the normal manner. The other sections consist of macro expansions, which supply information that is expanded in the appropriate places in the emulator source code. Each section may be used in more than one place in the source code, potentially expanding into totally different code each time. This is the main reason for using this technique: to automatically keep related fragments of code in sync after updating, where these fragments cannot be cleanly located together in the source files; usually a result of limitations in the language.

Figures 3.9 and 3.11 show the pre-processor meta-programming templates for instantiating the cell factories and registering the supported configuration names for each cell type, respectively. These appear in code that is internal to the emulator, that shouldn't need to be modified when new cell types are added. The corresponding expansions for these are shown in figures 3.10 and 3.12, respectively.

```
// Register the instruction cell factories, by creating singleton
// static instances of each.
#define INSTRUCTION_CELL(instruction_cell_class) \
    static Named_instruction_cell_factory<instruction_cell_class> \
        instruction_cell_class##_factory;
#include "cell-definitions.hh"  // Auto-generated file referencing
                                // all the cell header files.
#undef INSTRUCTION_CELL
```

Figure 3.9: Source code extract for auto-generating each cell type factory class, along with a file-static instance of it.

```
static Named_instruction_cell_factory<Add_cell> Add_cell_factory;
static Named_instruction_cell_factory<Mux_cell> Mux_cell_factory;
.
.
.
static Named_instruction_cell_factory<Reg_cell> Reg_cell_factory;
```

Figure 3.10: Example auto-generated source code resulting from the pre-processor meta-programming in figure 3.9.

The result of these features is to make adding a new cell type consist of simply adding a new header file describing the cell type, and recompiling the emulator. The resulting binary can then be used with an updated MDF and netlist which refer to the new cell type. It can even list usage information for the newly added cell types.

```
// Register the supported configurations for each cell type, by
// implementing the appropriate member function for each cell factory.
#define CONFIGURATION_NAMES(instruction_cell_class, configuration_list) \
    template<>                                                          \
    void                                                                \
    Named_instruction_cell_factory<instruction_cell_class>::            \
    register_configurations()                                           \
    {                                                                   \
        Name_for_index_map* names;                                      \
        names = supported_configurations_registry();                    \
        configuration_list                                              \
    }
#define CONFIGURATION_NAME(name) \
        (*names)[name] = #name;
#include "cell-definitions.hh"  // Auto-generated file referencing
                                // all the cell header files.
#undef CONFIGURATION_NAME
#undef CONFIGURATION_NAMES
```

Figure 3.11: Source code extract for auto-generating look-up tables associating a human readable name to each configuration word value, for each cell type.

```
template<>
void
Named_instruction_cell_factory<Add_cell>::register_configurations()
{
    Name_for_index_map* names;
    names = supported_configurations_registry();
    (*names)[ADD_ADD_SI] = "ADD_ADD_SI";
    (*names)[ADD_SUB_SI] = "ADD_SUB_SI";
    (*names)[ADD_ADD_HI] = "ADD_ADD_HI";
    (*names)[ADD_SUB_HI] = "ADD_SUB_HI";
    .
    .
    .
}

template<>
void
Named_instruction_cell_factory<Mux_cell>::register_configurations()
{
    Name_for_index_map* names;
    names = supported_configurations_registry();
    (*names)[MUX_SEL_IN1_IF_NEZ] = "MUX_SEL_IN1_IF_NEZ";
}

.
.
.

template<>
void
Named_instruction_cell_factory<Reg_cell>::register_configurations()
{
    Name_for_index_map* names;
    names = supported_configurations_registry();
}
```

Figure 3.12: Example auto-generated source code resulting from the pre-processor meta-programming in figure 3.11.

3.3.2 Contribution: Serialisation Algorithm

The serialisation algorithm is used at load-time to create the internal representation which drives the execution state machine. This internal representation imposes a significantly lower per-step overhead than serialising during execution.

The key requirement of this algorithm is to ensure that the result of executing the sequence of cell actions in the execution model, exactly matches the result of the original data flow graph for that configuration context. This simply requires that a cell's action (for calculating its output value) is scheduled before those of any dependent cells (*successors*). The serialisation algorithm requires extension to deal with situations where cells maintain internal state from one configuration context to the next. To explain this, we first give an example involving only combinatorial cells, then a second example showing the extension required to avoid apparent connection loops arising from internal state.

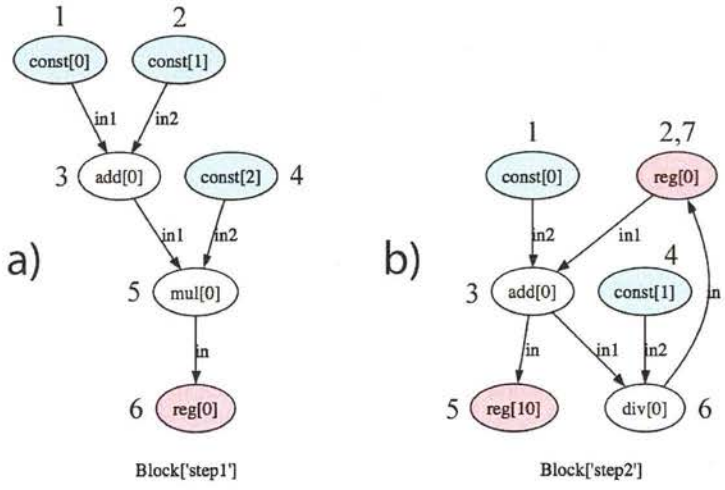


Figure 3.13: Example configuration contexts: (a) involving only combinatorial operations, (b) including a *connection loop*—this case is valid since the loop involves a register, which is a *terminal cell*.

The data flow graph example for a configuration context involving only combinatorial cells is given in figure 3.13(a). The operation of any purely combinatorial cell needs only an *operate* action to be defined. The constant cells supply the operands for a set of operations, and the final result is written to storage. A human might choose a sequence such as that shown by the numbers in figure 3.13(a). The algorithm employed by the emulator constructs the connection hierarchy between the active cells as a directed graph. Once the hierarchy is complete, the topological sort operation from graph theory is applied to the graph. The topological sort results in nodes being ordered in descending order of depth in the connection hierarchy. The ordered result is used to schedule the *operate* actions of each cell. Within a given depth, the cell actions could be scheduled in any order, without affecting the overall result. The direction of the arrows in figure 3.13 indicates the direction of data flow, and defines the terminology of *predecessor* feeding data to a *successor*, i.e. one of the successor's input ports is connected to the output port of the predecessor. In the completed hierarchy, a predecessor lies in some level lower than that of any of its successors.

Things are a bit more complicated than this, however, because some cells maintain internal state information. Taking registers as an example, the output of the cell does not depend on the input in the current configuration context; instead it depends on the internal state of the register cell.[4] This means that it is valid for a register to appear in a *connection loop*—where the output of the register is used in some sequence of operations, the result of which is stored back in the same register. This results in a cyclic graph, making a topological sort impossible. Essentially, the register cell can be thought of as two cells—one emitting the current value, and one receiving the new value. However, this is not a clean approach.

Alternatively, we can introduce the concept of *terminal cells*—i.e. cells where the inputs do not affect the outputs during the execution of that configuration context. Now, connection loops are valid if one of the cells in the loop is terminal. Terminal cells provide an *evaluate* action, in addition to an *operate* action. Calculating the output value of a terminal cell can always be done before anything else during the execution of a configuration context,[5] and writing to the input(s) of a terminal cell can always be done after anything else during the execution of a configuration context.[6] Furthermore, some cells need to have their state modified upon each configuration context transition (reconfiguration). This is done by providing an *update* action, that is performed once the rest of the actions have been executed. So, the algorithm is extended by scheduling all *evaluate* actions first, followed by the sequence of *operate* actions obtained from the topological sort, and finally all *update* actions are scheduled. Figure 3.13(b) shows an example, to which the algorithm would assign the sequence of cell actions given in figure 3.14.

```
const[0](evaluate), const[1](evaluate), reg[0](evaluate),
add[0](operate), div[0](operate), reg[0](operate), reg[10](operate),
reg[0](update), reg[10](update)
```

Figure 3.14: Cell action execution order for the example step DFG given in figure 3.13(b).

Registers are only a simple example of this problem. More complex examples include interfaces to streaming memories, and cells that are internally pipelined such that their output is delayed by (several) iterations. It has so far proven possible to map all supported cells to this mechanism, and this approach is quite effective in minimising the number of operations that need to be performed for each configuration context.

[4] which in turn usually depends on the input to the register from a previous configuration context.
[5] since the value does not depend on the result of any other cell during that configuration context.
[6] since the written value does not affect any other cells during that configuration context.

3.4 Results

The performance of the emulator was compared against a SystemC transaction-level model of the same instruction cell-based processor, and an FPGA implementation of the same array (i.e. a dynamic reconfigurable fabric on a static reconfigurable fabric). A quad 2.2GHz AMD Opteron PC was used as the host machine for the emulator and SystemC model. The FPGA used was the Virtex-4 LX 160.

Note that an FPGA implementation performs the same role as an HDL simulation of the processor architecture, and is used instead of an HDL simulation since it achieves much higher run-time performance, and so is much more suitable for the task of near real-time application demonstration.

3.4.1 Results: Execution Speed For a Range of Standard Benchmarks

The execution speed was used as the measure of performance. The reconfigurable array is intended to have a system clock of 500MHz. The maximum achievable clock on the FPGA implementation of the target processor is 12MHz[7]. The ratio of these gives the performance value for the FPGA.

For the other methods, the execution time was accurately measured and averaged over several runs. The averaging is necessary for user-space programs, in order to reduce random error introduced by pre-emptive context switches on the host. Execution speed is the time that the target application should have run for on the reconfigurable array, divided by the average run time on the model.

The following algorithms/applications were used:

- Discrete Cosine Transform (DCT) (for MPEG4/H.264 video).
- Finite Impulse Response (FIR) digital filter.
- Dhrystone (integer CPU performance bench-mark).
- MP3 (MPEG-1 layer 3) audio decoder (libmad).
- H.264 video decoder (ffmpeg).

The benchmarks were chosen to cover the realistic extremes of control-flow intensive and data-path intensive applications, whilst mapping to a core small enough to be implemented on the FPGA. Dhrystone is benchmark targeting traditional microprocessors, and aims to test their ability to process simple integer operations with lots of control flow. The DCT and FIR programs represent the opposite extreme—programs dominated by a single basic block with high core utilisation, which could easily be implemented in hardware. The MP3 and H.264 examples are real-world applications that make use of the DCT, but also have additional logic that leads

[7] determined by the critical path of the synthesised instruction cell array rendered on the FPGA, *which is the same irrespective of the target application.*

to control flow[8]. The purpose of this is to expose where the relative overheads lie between the different software-based simulation methods. The host-native performance figures are quoted to illustrate how well the applications match the capabilities of the host. The difference between the host-native performance and the performance of the emulator or System-C model can be attributed to two factors: the overhead of the simulation technique used, and the difference in quality of the optimising back-ends for the host-native compiler v.s. that for RICA.

	Emulator	SystemC model	FPGA model	Native
FIR	1.000	3.40e-3	0.52	21
DCT	1.000	5.52e-3	1.47	61
H.264	1.000	9.44e-3	1.43	59
MP3	1.000	12.00e-3	2.43	101
Dhrystone	1.000	76.00e-3	0.83	34

Table 3.1: Execution speed for various standard benchmarks, normalised to the speed of the emulator. The emulator is two orders of magnitude faster than the traditional SystemC model, and nearly as fast as an FPGA implementation of the target architecture. The overhead of emulation v.s. the overhead of SystemC's events is application dependent—the emulator is most advantaged for data-path intensive applications.

Table 3.1 shows that the performance of the emulator described in this thesis is good compared to the other simulation methods described. The real silicon (native) is between 21 and 101 times faster than the emulator, and the FPGA model is close in speed to the emulator. Since the FPGA is a model of the real silicon, it is a constant fraction of the speed of the real silicon. Both software models vary in execution speed (compared to the real silicon), depending on the application.

The relative performance of the emulator and SystemC model can also be seen to depend on the application. Since these two models use very similar cell implementations, written in C, this highlights the differences in the overheads incurred by the method of simulation. In addition to performing the actual work of the cells, the SystemC kernel incurs an overhead for each event generated by the active cells, and a further overhead at the end of each configuration context. The emulator on the other hand, only incurs the latter overhead, since everything except for the path of program execution is serialised prior to execution.

The Dhrystone example consists of many short basic blocks, which results in very low core utilisation. This represents the extreme of frequent configuration context changes with few cell operations in between. The FIR example represents the opposite extreme, where the program consists largely of one basic block, which results in very high core utilisation, and much core activity between configuration context switches. The results in table 3.1 show that the emulator is best advantaged when core utilisation is high, which supports this argument.

[8] thus making use of the core's ability to time domain multiplex its resources between different aspects of an application.

3.4.2 Results: Effect of Data Path Shape

To examine this variance in relative performance (execution speed) between the emulator and the SystemC model with different programs, some small test programs were written. Each consists of a single loop mapped into a single configuration context. In each case, the loop body consists of a relatively simple sequence of arithmetic operations to apply to each member of a data set. The programs differ in when the operations for a given member of the data set are executed. The programs are visualised in figure 3.15.

	Total operations per iteration	Critical path	Relative speed
Parallel	26	16ns (5 operations)	1246x
Combinatorial	26	40ns (9 operations)	1377x
Sequential	11	16ns (5 operations)	2258x

Table 3.2: Complexity and relative execution speed (emulator v.s. SystemC model) for some simple test programs written to investigate the reason for the application-dependent relative execution speed. Execution time for the emulator depends only on the number and type of operations present, and not their order. The SystemC model however is affected by data path dependencies (demonstrated by 'Parallel' v.s. 'Combinatorial'). The SystemC model incurs a higher per-step/iteration overhead (shown by the degraded relative performance in the 'Sequential' example).

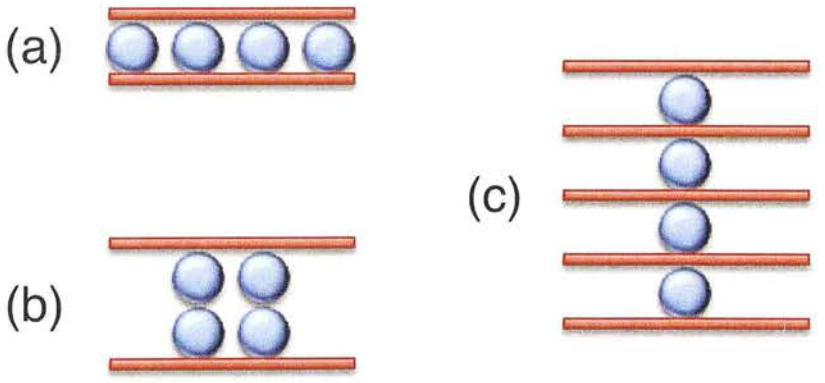


Figure 3.15: Visual representation of the kernels used in table 3.2. Red lines show configuration context boundaries, and the blue circles represent the data path that is replicated. Time runs vertically. (a) Parallel: the four copies of the data path all run in parallel in the same configuration context. (b) Combinatorial: two copies of the data path are chained together, and two copies of this macro are executed in parallel in the same configuration context. (c) Sequential: the configuration context contains only one copy of the data path, and so has to complete four times as many iterations to process the same amount of data.

To test the effect of the number of events generated per iteration, one program (`Parallel`—figure 3.15(a)) performs the operations of four members of the data set in parallel; whilst another program (`Combinatorial`—figure 3.15(b)) also operates on four members of the data set per iteration, but a data dependency exists preventing them from running entirely in parallel

(however they still overlap to a certain extent). The number and type of operations performed per iteration in both of these programs is the same; however the latter (`Combinatorial`) case has a longer critical path. The relative performance of the emulator and SystemC model is similar for both programs, the results of which are shown in table 3.2.

The execution time of the emulator should depend only on the operations performed, and not the order. For the SystemC model, the longer critical path (and number of operation chained together) causes more flutter as the combinatorial paths stabilise, resulting in more transition events being generated. However, the execution time for each event is very small compared to the time taken to schedule the events, and the results in fact show a slight relative gain. This indicates that the run-time scheduling is easier when the timing of the events is more sequential.

To test the effect of the number of operations per iteration, another program was written (`Sequential`—figure 3.15(c)), this time with only one member of the data set operated on per iteration of the kernel. This requires that four times as many iterations are performed. A significant increase in the relative speed of the emulator can be seen compared to the previous test programs. This therefore indicates that the SystemC model incurs a disproportionately large overhead per iteration, which supports the earlier observation with the standard benchmarks.

The source code to all three programs can be found in appendix A, along with the data flow graphs of the resulting configuration contexts (only the kernel configuration contexts are shown).

3.5 Summary

This chapter presented algorithms and methodologies used to implement a high-speed simulator (emulator) for the RICA architecture. Such a simulator is implemented entirely in software. In order to simulate a data path architecture on a conventional microprocessor, the data paths must be broken up into an equivalent sequence of operations on the microprocessor (i.e. they must be *serialised*).

Traditional simulation methodologies for data path machines were described—HDL simulators, and their derivatives. These can be used to simulate RICA, but since RICA is intended to be reconfigured very frequently during normal operation, simulations tend to be very slow. This is because the serialisation is performed by the simulator upon every iteration of every configuration context.

The approach proposed in this thesis takes advantage of knowledge about how RICA changes state, which allows the serialisation to be performed in advance—before running the program. This moves the overhead from run-time to load-time, and is a constant cost amortised over the entire execution time of the application. This makes it particularly advantageous for long-running programs, where execution time is also most significant.

The emulator operates on the set of configuration contexts that describe a given program. The next chapters look at other aspects of the tool chain: algorithms and methodologies for creating those configuration contexts, and maximising their performance. The emulator can be used as an integral part of these tools, for use in feedback-directed optimisation.

Chapter 4
Scheduling

The overall design premise in this work is to use C source code to program a coarse-grained reconfigurable computing architecture. An important feature is the ability to use dynamic reconfiguration to time division multiplex sections of a design larger than the target array. The tool chain to do this has been partitioned into separate tools: a compiler, a scheduler, and a routing tool.

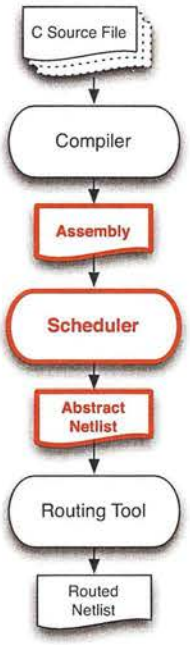


Figure 4.1: Process of converting C source files into a set of configuration contexts for the target reconfigurable array, partitioned into a tool chain. The tool and files relevant to this section are highlighted.

The reason for this partitioning is as follows: in order to leverage the wealth of existing compiler technology and optimisation passes, an existing compiler was used (GCC [73]). GCC is designed to be generic; the extent of this generality adequately covers conventional computing architectures, and others have shown how it can also be extended to less conventional architectures [74, 75]. However, the nature of coarse-grained reconfigurable computing architectures breaks too many of the assumptions inherent in this framework, and so is difficult to capture. This results in poor core utilisation. So, the compiler is used to generate an intermediate representation that a stand-alone tool can then work on, to better extract the available parallelism. This stand-alone tool is the scheduler.

Finally, routing is done separately since the search space is very large, and so the process is rather time consuming. Routing needs to be done in order to program the real device, however the behaviour and approximate timing can be analysed without having to perform routing. The time saved by only performing routing when needed, drastically improves the iteration rate of the design cycle, which makes working with the target architecture very similar to working with a conventional microprocessor, rather than what is common with reconfigurable hardware (using HDL).

4.1 Problem Description

The intermediate representation taken from the compiler is in the form of a serial instruction stream. This instruction stream is to be mapped onto a core that has the potential for large amounts of instruction-level parallelism. Each instruction represents the operation of a particular instruction cell in the array. The instructions are grouped into basic blocks, inside which there is no conditional flow control between instructions. An ideal schedule for each basic block of this instruction stream will consist of each independent data path in a given basic block executing in parallel, such that the critical path of the schedule as a whole is that of the longest data path. Deviations from this ideal will be necessary if insufficient resources are available in the core or if certain timing constraints can't be met, in which case the goal is to minimise the increase in total latency, whilst still executing all of the data paths. This allows designs (much) larger than the core itself to be executed.

The task of scheduling is complicated by the presence of a paradox: scheduling requires the calculation of each data path's critical path length (delay). However, this critical path length depends on the length of interconnect used to connect each cell in the data path together. This cannot be known until after scheduling and routing have been performed, leading to infinite regression (c.f. which came first, the chicken or the egg?). This can be partially avoided by making a simplification: the scheduling estimates the delay of the interconnect. This estimate is based on an empirically obtained average interconnect length[1] multiplied by the measured delay of a path segment and associated s-box. This average is obtained by analysing the fully routed configuration contexts (*steps*) of a statistically significant set of programs mapped onto a given array. This is then given as a property of the target array. Later, more accurate timings can be calculated from the routed netlist, and timing-sensitive configuration values adjusted if required (i.e. the *RRC* fields).

The data model was designed to be operation-centric. The interdependencies between the operations are captured in two ways: direct via connections, and indirect via constraints. Physical registers are represented as operations, and there are several types of register operation: input register, output register, temporary register, and pipeline stage register. The operations do not directly correlate to cell configurations. This is because the task of scheduling is largely about inferring wires and registers from what appears in the assembly[2], and defining additional register usage[3]. This operation-centric model has to be transferred to a cell-centric model for netlist generation.

[1] in terms of number of path segments.
[2] which mainly uses registers as the interconnect agents.
[3] temporary registers and pipeline stage registers.

4.1.0.1 Aims

To produce a scheduling tool that allows the assembly produced by a compiler to be efficiently packed into configuration contexts for programming a dynamically reconfigurable array. More specifically:

Correctness: To construct a schedule of configuration contexts that run sequentially to perform the intended functionality of any given basic block. The schedule must adhere to the available resources in the target core, and the resulting state change after executing the schedule must match that implied by the assembly after having executed all the instructions in the basic block. It must also obey a set of other architecture-specific criteria, such as maximum representable step time (RRC field overflow).

Efficiency: The resulting schedule should consist of as few configuration contexts as possible (to minimise program memory overhead), and the total of the context critical paths should be as small as possible (to make it as fast as possible). This therefore involves attempting to parallelise the data paths of the data flow graph.

Furthermore, by mapping loops to individual configuration contexts, pipelining can be applied to dramatically improve throughput (discussed in chapter 5). Therefore, the work on scheduling can be viewed as being for the purpose of generating basic blocks that are good candidates for pipelining.

4.1.0.2 Objectives

- Devise a data model that can describe a wide range of target architectures, in a manner that allows for easy static analysis.
- Devise a series of algorithms that operate on this data model, to transform basic blocks into valid configuration contexts.

4.1.0.3 Novelty

List scheduling is extended to improve the ability to pack data paths into as few a steps as possible, in an algorithm called a *Tree follower* (section 4.9). This comprises a new layer built on top of list scheduling, which can dynamically re-order the ready list in order to give precedence to operations that lie on the current data path (or arm of that data path).

As a side effect of this packing, more data paths can become split across step boundaries, requiring registers to store the values of the broken connections over each step boundary. For cores with a very limited number of registers, this can lead to register starvation. A series of algorithms were devised to avoid this—*Register starvation avoidance* (section 4.10).

Furthermore, a series of optimisation and analysis passes are presented that improve the scheduling efficiency—*Live register identification* (section 4.7), and aid the routing tool to achieve a more optimum allocation—*Global live register reallocation* (section 4.12), which improves routability and reduces combinatorial delay, thus improving throughput.

4.2 Example

To illustrate the purpose of the various algorithms, a simple example assembly is presented here in figure 4.2, which is to be executed on an array with the resource counts given in table 4.1. Both the array and the basic block have been chosen to be very much smaller than what would be typical, for the purpose of making it easier to comprehend.

The effect of the key stages of scheduling, from assembly to abstract netlist, will be demonstrated. At the end of this process, an abstract netlist will be obtained. The example is then taken slightly further, to illustrate how the netlist could be mapped onto the array, following allocation & routing.

```
block1:
    ADD       out = r0     in1 = r1     in2 = r2     conf = 'ADD_SUB_SI
    CONST     out = r3                               conf = 4
    SHIFT     out = r0     in1 = r0     in2 = r3     conf = 'SHIFT_SLL_SI
    CONST     out = r4                               conf = 1
    ADD       out = r5     in1 = r5     in2 = r4     conf = 'ADD_ADD_SI
    CONST     out = r2                               conf = 3
    MUL       out = r7     in1 = r2     in2 = r7     conf = 'MUL_MUL_SI
    MUL       out = r8     in1 = r8     in2 = r7     conf = 'MUL_MUL_SI
    CONST     out = r4                               conf = 5
    ADD       out = r6     in1 = r6     in2 = r4     conf = 'ADD_ADD_SI
    MOV       out = r4     in  = r7
    MOV       out = r2     in  = r8
```

Figure 4.2: Example assembly for a basic block. This example contains 4 independent data paths.

Cell type	Instance count
add	2
const	3
mul	1
shift	1
reg	9

Table 4.1: Available instruction cell resource count for a hypothetical, artificially small RICA array.

Performing data flow graph (DFG) analysis on the basic block, we can determine the connectivity between the various operations. The DFG is flattened—intermediate registers are replaced with wires. The only registers that remain in this data model are those that bring values into the basic block (termed *input registers*), and those that bring values out of the basic block (termed *output registers*).

The flattened DFG is shown in figure 4.3. A single basic block may consist of several completely independent data paths, as can be seen in this example. To illustrate where they come from, figure 4.4 shows the previous assembly with white space and comments inserted.

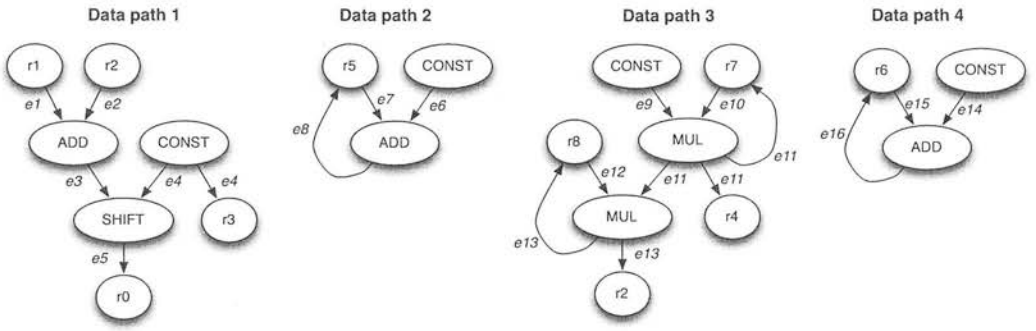


Figure 4.3: Data flow graph (DFG) extracted from the assembly in figure 4.2.

```
// Data path 1:
ADD         out = r0      in1 = r1      in2 = r2      conf = 'ADD_SUB_SI
CONST       out = r3                                  conf = 4
SHIFT       out = r0      in1 = r0      in2 = r3      conf = 'SHIFT_SLL_SI

// Data path 2:
CONST       out = r4                                  conf = 1
ADD         out = r5      in1 = r5      in2 = r4      conf = 'ADD_ADD_SI

// Data path 3:
CONST       out = r2                                  conf = 3
MUL         out = r7      in1 = r2      in2 = r7      conf = 'MUL_MUL_SI
MUL         out = r8      in1 = r8      in2 = r7      conf = 'MUL_MUL_SI

// Data path 4:
CONST       out = r4                                  conf = 5
ADD         out = r6      in1 = r6      in2 = r4      conf = 'ADD_ADD_SI

// Data path 3 (continued):
MOV         out = r4      in  = r7
MOV         out = r2      in  = r8
```

Figure 4.4: Assembly instructions from figure 4.2 grouped by which independent data path they belong to.

Looking at just the instructions from this basic block, a first approximation to the input and output registers can be made:

Input registers: r1, r2, r5, r6, r7, r8

Output registers: r0, r1, r2, r3, r4, r5, r6, r7, r8

The nodes of the data flow graph are registers or operations, each corresponding to a physical instruction cell in the array. The edges of the data flow graph represent pieces of information (values) passed between the nodes. Table 4.2 shows the edges of the data flow graph, and the operations that create their value.

Edge	Value represented	Output registers
e1	input from r1	r1
e2	input from r2	-
e3	result of: r0 ← r1 ADD r2	-
e4	result of: r3 ← CONST 4	r3
e5	result of: r0 ← r0 SHIFT r3	r0
e6	result of: r4 ← CONST 1	-
e7	input from r5	-
e8	result of: r5 ← r5 ADD r4	r5
e9	result of: r2 ← CONST 3	-
e10	input from r7	-
e11	result of: r7 ← r2 MUL r7	r4, r7
e12	input from r8	-
e13	result of: r8 ← r8 MUL r7	r2, r8
e14	result of: r4 ← CONST 5	-
e15	input from r6	-
e16	result of: r6 ← r6 ADD r4	r6

Table 4.2: All edges from the example data flow graph in figure 4.3, and the corresponding assembly in figure 4.2.

If the entire data flow graph can fit on the array at once, then all the edges become wires[4]. Otherwise, the data flow graph has to be split into fragments, each of which are small enough to fit on the array, and the fragments are executed sequentially.[5] Any individual data paths that are too big to fit on the array in one step must be split. The edges that are split must be replaced by physical registers in the core, to store the temporary value, which is then read back in some later step when the array is reconfigured with the remainder of the data path. Registers used for this purpose are called *temporary registers*. The scheduling algorithm is responsible for choosing the best places to split large data paths, and how the fragments of different data paths are packed together into steps. Figure 4.5 shows the resulting schedule for the example basic block.

[4] the interconnect in the core.

[5] time division multiplexing the array resources.

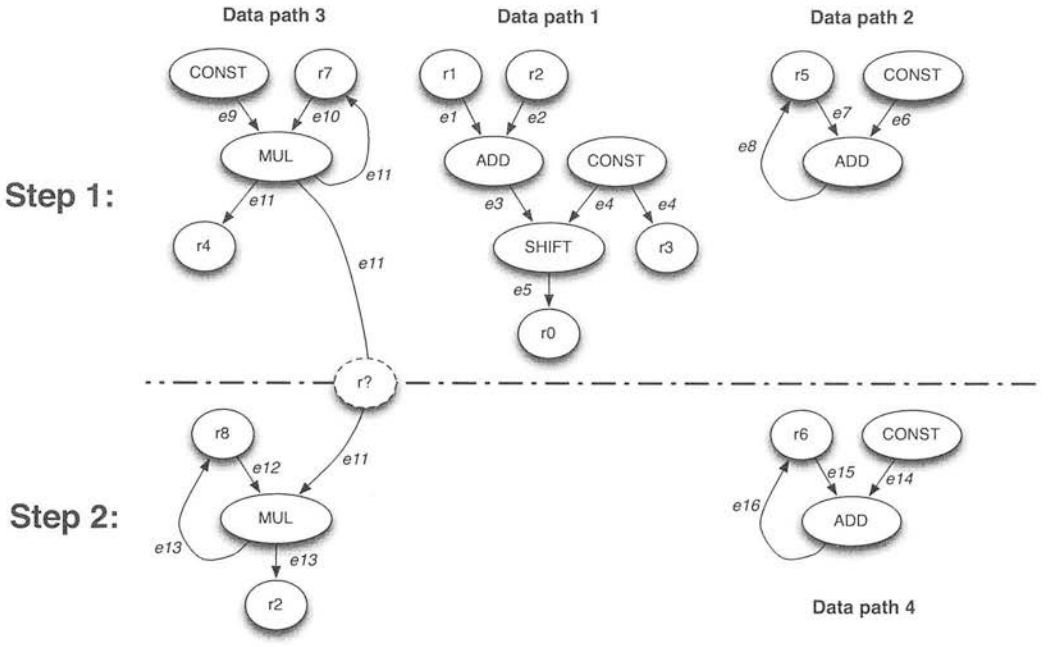


Figure 4.5: Data flow graph from figure 4.3 scheduled for the example array defined in table 4.1. There are insufficient cells available for this basic block to become a single step, so it has been split. Data path 3 is spread across both steps, requiring a temporary register (dotted outline) to transport the value of the broken edge across the step boundary.

Across each step boundary, the scheduler must choose which registers to use for storing the temporary results. In our example, all the registers are active in the basic block. This means that there are no registers available purely for use as temporaries. As a result, the scheduler must try to re-use the active registers for this purpose. Since the steps corresponding to a given basic block are always executed in sequence, the state of the registers needs only to be preserved on entry to the first step, and on exit from the last step. This allows the scheduler to use any of the active registers in any way that it likes across any step boundary internal to this basic block. As a result, the scheduler considers all values as temporaries across each internal step boundary, as shown in figure 4.6. For each step boundary, only a single register needs to be assigned to store each edge where the producer and any consumer of that edge lie on opposite sides of that boundary. Therefore, duplicates only need to be stored once, e.g. $e11$ in the example needs only one temporary register to bring it into the second step, despite the edge needing to be stored in two output registers.

Registers are assigned to temporaries across each internal step boundary from the available pool. This pool consists of all the active registers, plus any additional registers that are known to be dead. Once all operations that read the value brought in by an input register have been executed, that value no longer needs to be stored (it becomes dead). Similarly, the value brought out by an output register only needs to be stored once the operation creating that result has been executed. This leaves some of the registers free for storing other internal edges across internal step boundaries. On exit from the last step, the register names as given in the assembly are honoured again.

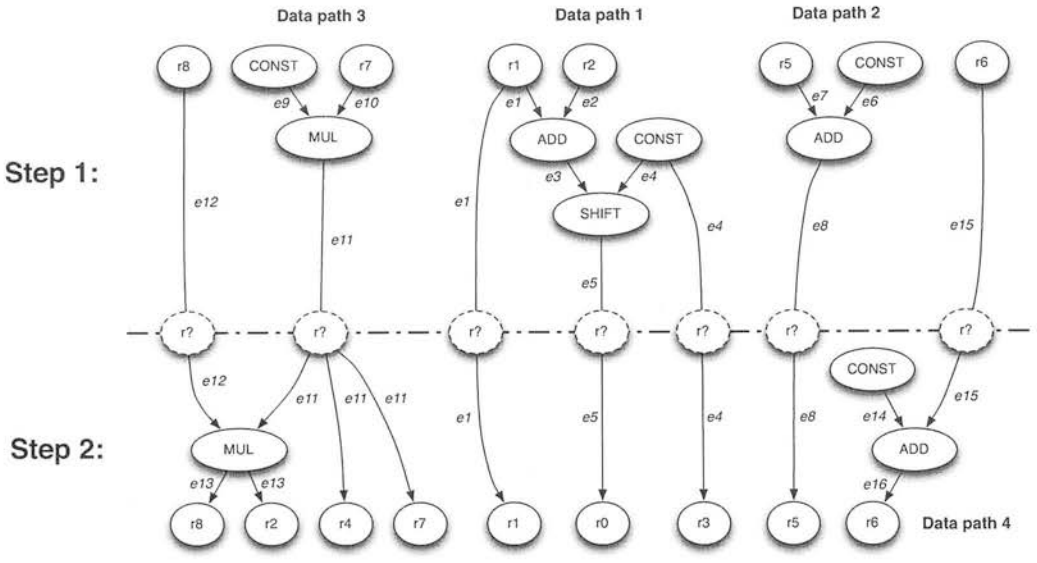


Figure 4.6: Example schedule from figure 4.5 showing all the temporary registers (shown by a dotted outline) that are needed to bridge values across the internal step boundary, as seen by the scheduling algorithm.

The size of the available pool of registers is crucial to how well the scheduler can parallelise the code. If insufficient registers are available to store the temporary registers over any given step boundary, then a new schedule has to be constructed (with relaxed timing), with less parallelism. In many (if not most) situations, the active registers alone are insufficient for this purpose. Inactive registers cannot safely be used, as they could be storing information across this basic block for use later in the program (termed *dormant registers*). A simple solution to this is to reserve a fixed set of registers entirely for the purpose of storing temporaries—termed *scratch registers*.

An alternative approach is to traverse the entire control flow graph (CFG) of the program, tracking the possible uses of each register. This information describes which inactive registers are dormant. All other inactive registers are free to use for storing temporaries. Furthermore, it is possible to determine which of the active registers store useful information on exit from the basic block. All others therefore become free for use as temporaries on exit from the step that last reads their value. Without this information, all active registers must be considered live on exit from the basic block. This control flow graph traversal process is called *live register identification* (section 4.7). It is not used in this example, since the other basic blocks in this program are not shown.

The schedule with registers allocated between each step, can then be written in the form of an abstract netlist, as shown in figure 4.7, augmented with timing information etc. derived from the per-step data flow graphs. This marks the end of the tasks performed by the scheduler. The abstract netlist is sufficient to execute the program on a high-level (functional) simulator of the array.

```
sequence
{
    .
    .
    .

    step[%block1]
    {
        reg[1];                                 // e1(in)
        reg[2];                                 // e2(in)
        add[0]    { in1=reg[1].out;
                    in2=reg[2].out;
                    conf='ADD_SUB_SI; }         // e3
        const[0].conf=4;                        // e4
        shift[0] { in1=add[0].out;
                    in2=const[0].out;
                    conf='SHIFT_SLL_SI; }       // e5
        reg[3].in=const[0].out;                 // e4(out)
        reg[0].in=shift[0].out;                 // e5(out)
        const[1].conf=1;                        // e6
        reg[5].in=add[1].out;                   // e7(in), e8(out)
        add[1]    { in1=reg[5].out;
                    in2=const[1].out;
                    conf='ADD_ADD_SI; }         // e8
        const[2].conf=3;                        // e9
        reg[7].in=mul[0].out;                   // e10(in), e11(out)
        mul[0]    { in1=const[2].out;
                    in2=reg[7].out;
                    conf='MUL_MUL_SI; }         // e11
        reg[4].in=mul[0].out;                   // e11(out)
        rrc.conf=4'b0001;                       // Critical path: 8.1ns
    }
    step[%block1_Step2]
    {
        reg[4];                                 // e11(in)
        reg[8].in=mul[0].out;                   // e12(in), e13(out)
        mul[0]    { in1=reg[8].out;
                    in2=reg[4].out;
                    conf='MUL_MUL_SI; }         // e13
        const[0].conf=5;                        // e14
        reg[6].in=add[0].out;                   // e15(in), e16(out)
        add[0]    { in1=reg[6].out;
                    in2=const[0].out;
                    conf='ADD_ADD_SI; }         // e16

        rrc.conf=4'b0001;                       // Critical path: 8.1ns
    }

    .
    .
    .
}
```

Figure 4.7: The abstract netlist resulting from the schedule shown in figure 4.5.

To get more accurate timing results, and to be able to program a physical array, the configuration of the interconnects needs to be defined. Continuing the example to this end, we could run the abstract netlist through the allocation and routing tool, to produce a fully qualified netlist. This constructs the paths between the active cells according to the connectivity defined in the abstract netlist. The various operations can be re-allocated to different instances of cells of the same type, that are closer together.

The fully qualified netlist can be directly converted into a configuration bitstream for the physical hardware. The physical configurations would look something like those shown in figures 4.8 and 4.9. For the purposes of illustration, the allocation in these examples is the same as in the abstract netlist.

Resource re-allocation can only be done safely for cells that have no state. Normally, registers should be non-relocatable, because they maintain state. Since the scheduler is responsible for allocating registers, and since it is not routing aware,[6] the registers that it chooses are arbitrary. As a result, this causes serious routability problems. However, by using the global register reallocation information created by the scheduler, the allocator can reallocate registers, drastically improving the situation. Allocation & routing, and bitstream generation are outside the scope of this thesis.

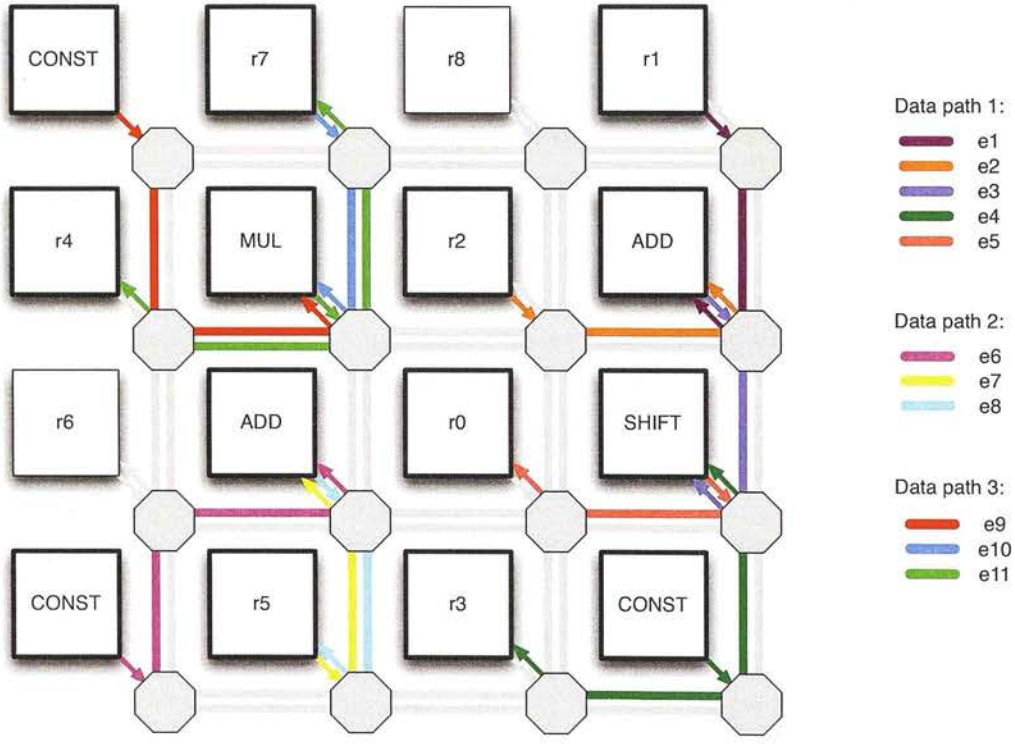


Figure 4.8: The first step of basic block `block1` (label `block1`), mapped onto the array.

[6] and it would be distinctly non-trivial to make it so.

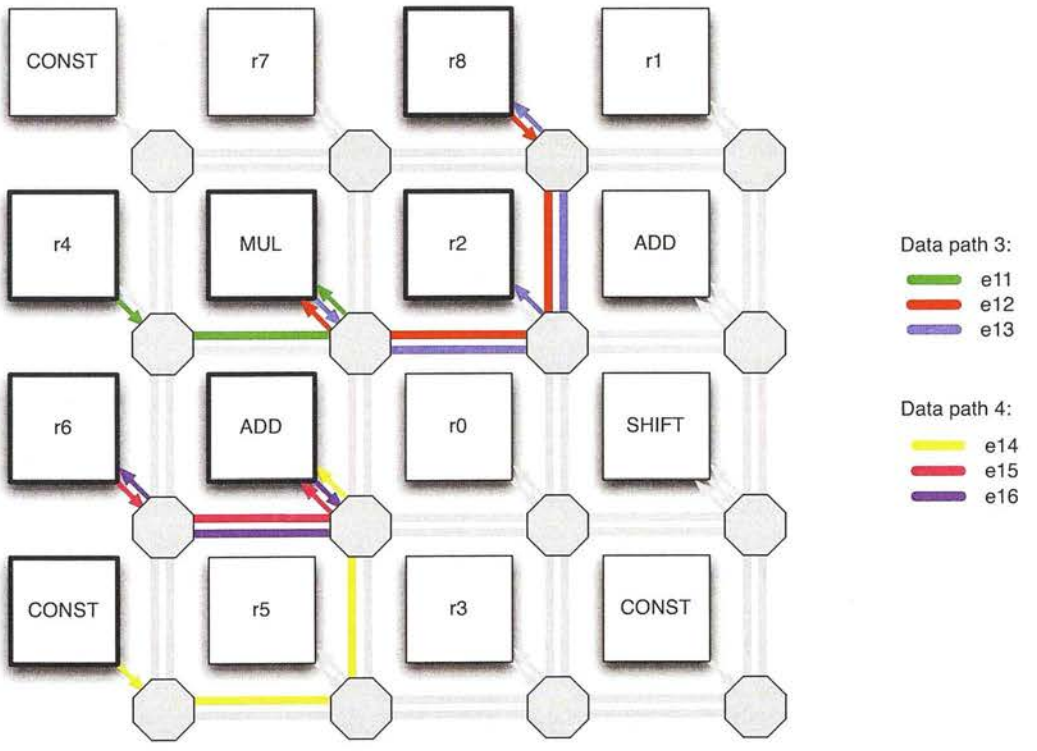


Figure 4.9: The second step of basic block `block1` (label `block1_Step2`), mapped onto the array.

4.3 Scheduling Stages Overview

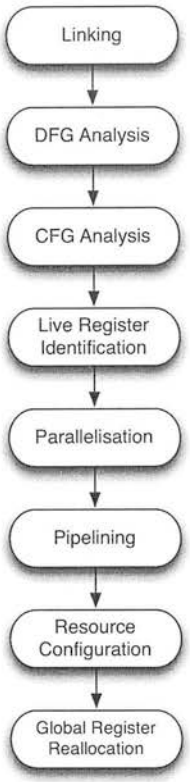


Figure 4.10: The tasks performed by the scheduler—stages to convert from assembly to abstract netlist.

The task of converting from assembly to abstract netlist is broken into the following stages:

Linking: Multiple assemblies (from user code and system libraries) are merged together to form the complete program, and dead code is identified and stripped away. Data symbols are allocated to physical addresses in data memory. Section 4.4.

DFG analysis: An internal representation is created for each basic block of the program, where the list of assembly instructions is converted into a data flow graph (DFG). Registers in the assembly become wires, except where they bring a value into the basic block (termed *input registers*), or a value out of the basic block (termed *output registers*). Section 4.5.

CFG analysis: The list of possible jump targets (basic blocks) is determined for each basic block in the program. Section 4.6.

Live register identification: The information derived from the CFG is used to determine which of the input and output registers in each basic block actually bring useful data into and out of each basic block. This helps work around the main drawback of operating from assembly: the assembly provides no way of explicitly marking a register as no longer containing useful information, and thus available for storing another value. Section 4.7.

Parallelisation: Each basic block is scheduled into a series of one or more steps (configuration contexts), by mapping the data flow graphs onto cells in the array. Constraints identified in DFG analysis must be adhered to, and the live register information extracted from CFG analysis is used to maximise the number of registers available for use as *temporary registers* to bridge partial results across step boundaries where data paths have to be split. This is the key area of the scheduler. A brief overview of this stage is given in section 4.8, however a full description is presented in a section on its own—section 4.9.

Pipelining: The parallelised data paths can be pipelined at this stage, although it is best performed on a routed netlist, where the interconnect delays of each individual path are known. That is the scope of chapter 5.

Resource configuration: The schedule is written out in the form of a netlist, which involves some additional special-case logic to complete the configuration, such as calculating step duration, memory access timing field generation, and memory cascading.

Global register reallocation information: After scheduling is complete, the roles of all the registers have been decided, so it is then possible to track the flow of data through all registers throughout all paths of possible control flow in the entire program. This information is then written to the netlist, to allow the routing tool to freely reallocate the registers in order to improve routing efficiency. Section 4.12.

The linking and CFG analysis stages are similar to those found in the DIABLO optimising linker framework [76], although they were developed independently. The work here adds more detailed live register identification, which is unnecessary with traditional microprocessors, but very important for data path architectures. Work published on DIABLO gives an idea of the sorts of optimisations that are possible when working at this level. Some of these optimisations (such as basic block merging, conditional branch merging, constant propagation, and constant replication) were implemented for RICA, but are outside the scope of this thesis.

4.4 Linking

In conventional C tool chains, the task of creating an executable binary is split into two stages: compilation and linking. Compilation consists of converting C code from a compilation unit[7] into assembly. This assembly is then assembled, to create a binary object file. Multiple object files are then linked together to form the complete executable. This two-layer process has the advantage of introducing scalability—the task of compilation requires much more processing time and memory than linking, so being able to split an arbitrarily large program into multiple pieces, each compiled individually (termed *incremental compilation*), allows the memory requirement to be limited to that of a single assembly.

The compiler assigns globally unique symbolic names to each function, each basic block, and each global data symbol[8] in the program. It does not bind physical (static) addresses to these; that is the task of the linker.

The differences between the target reconfigurable architecture compared to regular microprocessors become apparent when deciding upon a binary object file format. The task of preparing a program for execution on a regular microprocessor, from the assembly output of the compiler, is simple—the assembly mnemonics directly translate into binary instruction patterns, and the hardware is designed to execute sequences of these. However, the task of preparing a program for execution on the target architecture—i.e. the tasks of scheduling, allocating, and routing—is non-trivial, and not practical to perform at run-time. Furthermore, the combination of this and the Harvard-style memory architecture—where the program memory is physically separate to the data memory, and is strictly read-only from the perspective of the target device—makes dynamically generated code impractical. Also, dynamic linking offers little advantage in the sort of embedded applications that are targeted. This makes it difficult to use existing object file formats, and the features that they provide are unnecessary and/or inappropriate.

As a result, a custom binary object file format was developed, that shares more in common with the hardware world—a netlist. The concept of a netlist was extended to represent multiple configuration contexts, and given ways to represent the initialisation values for certain areas of memory. The scheduler tool takes in assembly files and performs the role of an optimising linker, to create a single netlist which is then passed into the hardware-domain tools (mapper and bitstream generation).

Figure 4.11 shows an example of the assembly format emitted by the GCC compiler RICA back-end, with the important features highlighted. Naming conventions are used for each symbol type, to make lexical scanning easier.

So, the conventional tasks of a linker are performed: the assembly is analysed, and the live functions and data symbols are identified. Dead functions and data symbols are stripped. Live data symbols are statically bound to fixed addresses in data memory, and a symbol table is constructed to describe this mapping.

[7] a C source file plus any headers it includes.

[8] resulting from global and static variables in C.

```
.section program_rom    // Code.

.align 4
.global _main
.proc _main    // Beginning of function '_main'.
_main:
    CONST   out= r12      conf= -12
    ADD     out= r13      in1= r1             in2= r12     conf= 'ADD_SI
    WMEM    in= r2        in_addr= r13        in_off= 4    conf= 'WMEM_SI
    ...
    CONST   out= w33      conf= @L404   // Absolute address of another block.
    JUMP    in_addr= w33  conf= 'JUMP_ALWAYS
L11:
    // Accessing the 4th entry of a global array.
    RMEM    out= r3       in_addr= !_g_matrix  in_off= 12   conf= 'RMEM_SI
    MOV     out= r23      in= r25
    MOV     out= r22      in= r3
    ...
L466
    RMEM    out= r9       in_addr= r1          in_off= 0    conf= 'RMEM_SI
    RMEM    out= r2       in_addr= r1          in_off= 4    conf= 'RMEM_SI
    ADD     out= r1       in1= r1              in2= 32      conf= 'ADD_SI
    JUMP    in_addr= r9   conf= 'JUMP_ALWAYS    // Return from function.
.endproc    // End of function '_main'.


.section data_ram    // Initialised read/write globals and static locals.

.align 4
.global _g_matrix    // Global data symbol (3x3 array of 32-bit values).
_g_matrix:
    .long    1
    .long    0
    .long    0
    .long    0
    .long    1
    .long    0
    .long    0
    .long    0
    .long    1
.align 4
.global _g_out        // Global array initialised to zeroes.
_g_out:
    .space    64
```

Figure 4.11: Example RICA assembly with the `main` function (showing a few of its basic blocks), and some global data symbols.

Parallelisation (section 4.8) is performed on the basic blocks, to create configuration contexts (steps), which are the fundamental unit in the program memory. Note that the step names remain symbolic in the netlist, and are bound to physical addresses during bitstream generation. This is because the size of each step may not be the same, e.g. if program stream compression is used, and the resulting size cannot be determined until after mapping is complete.

4.4.1 Live Symbol Identification Algorithm

After merging the assemblies into a flat namespace, linking begins by identifying the program entry point (boot strap). This is established by means of a naming convention—a function with a particular name is looked for. The program entry point is initially considered to be live.

Then the following iterative process begins: the functions that were newly identified as live are scanned. This involves looking through each instruction of each basic block within that function. The operands of the instruction are scanned for literals of particular types (determined by a naming convention):

Basic block labels: These are identified by the lexical scanner by the '`@`' or '`&`' prefix (for relative and absolute addressing modes, respectively). These are assumed to represent the direct target of the jump at the end of the current basic block, or if subsequently stored into data memory, represent a function pointer—i.e. a potential jump target for any indirect jumps anywhere in the program. If any such label matches the name of a function, then that function is considered to be live.[9]

Data symbol labels: These are identified by the lexical scanner by the '`!`' prefix. This marks that symbol as having been referenced, and therefore live. Only live data symbols are assigned physical addresses and included in the symbol table.

The process ends when no new live functions were discovered during the last pass.

This algorithm is low-cost, and the rather coarse information identified here is sufficient for the intended purposes. Since the flattened assembly can be quite large,[10] this is an important consideration in order to maintain scalability. The later stages (DFG Analysis and CFG Analysis—sections 4.5 and 4.6) refine this information, by which stage they are operating on a much smaller amount of data (i.e. mostly only live code).

There is room for optimisation in the assignment of physical addresses to data symbols: alignment and locality of reference can have a significant effect on the throughput of accessing that memory. Memory access patterns can be analysed and feedback-derived optimisation can be applied to relocate data symbols to achieve a more optimal physical address assignment. However, this has not yet been explored in this work.

[9] as the first basic block of each function is named after the function, by convention.
[10] containing implementations of all the system libraries as well as all the user code.

4.5 DFG Analysis

The compiler describes a program in terms of *basic blocks*, which are the fundamental unit of control flow. The instructions within a basic block are intended to be executed in sequence, without interruption (branching). A basic block ends either by passing control directly to the next basic block in sequence, or by jumping to another basic block. The choice between these can be conditional, i.e. the choice of which basic block to execute next can depend on values in the data paths, and thus depends on the current state of the machine.

The uninterruptible nature of a basic block makes them effectively describe a fixed circuit consisting of one or more data paths. The *data flow graph* (DFG) is a graphical representation of these data paths. These circuit descriptions can be used to generate configuration contexts for the reconfigurable core. Each instruction represents an active cell in the core, and the registers used as operands and results in the assembly describe the connections (wires) between these cells. Connections that have no start point or end point indicate that these values come directly from/to physical register cells in the core.

```
block1:
    ADD       out = r0     in1 = r1     in2 = r2     conf = 'ADD_SUB_SI
    CONST     out = r3                               conf = 4
    SHIFT     out = r0     in1 = r0     in2 = r3     conf = 'SHIFT_SLL_SI
    CONST     out = r4                               conf = 1
    ADD       out = r5     in1 = r5     in2 = r4     conf = 'ADD_ADD_SI
    CONST     out = r2                               conf = 3
    MUL       out = r7     in1 = r2     in2 = r7     conf = 'MUL_MUL_SI
    MUL       out = r8     in1 = r8     in2 = r7     conf = 'MUL_MUL_SI
    CONST     out = r4                               conf = 5
    ADD       out = r6     in1 = r6     in2 = r4     conf = 'ADD_ADD_SI
    MOV       out = r4     in  = r7
    MOV       out = r2     in  = r8
```

Figure 4.12: Example assembly for a basic block. This example contains 4 independent data paths.

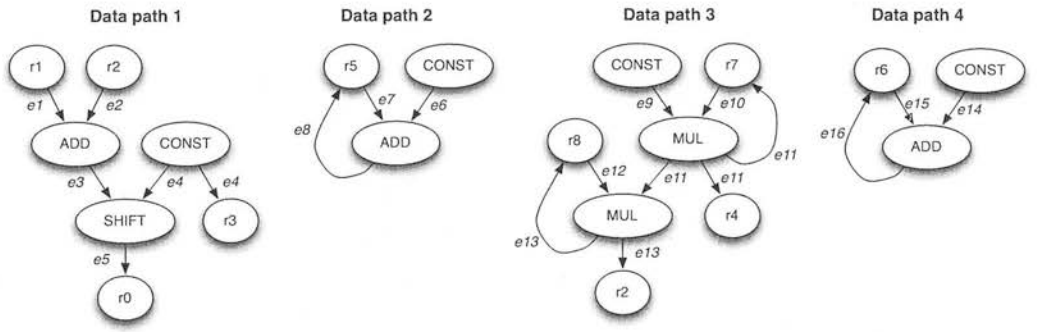


Figure 4.13: Data flow graph (DFG) extracted from the assembly in figure 4.12.

Figure 4.12 shows an example basic block, in which analysis reveals four independent data paths, as shown in figure 4.13. Figure 4.14 reformats the assembly to show where these data paths come from.

```
// Data path 1:
ADD         out = r0     in1 = r1     in2 = r2     conf = 'ADD_SUB_SI
CONST       out = r3                               conf = 4
SHIFT       out = r0     in1 = r0     in2 = r3     conf = 'SHIFT_SLL_SI

// Data path 2:
CONST       out = r4                               conf = 1
ADD         out = r5     in1 = r5     in2 = r4     conf = 'ADD_ADD_SI

// Data path 3:
CONST       out = r2                               conf = 3
MUL         out = r7     in1 = r2     in2 = r7     conf = 'MUL_MUL_SI
MUL         out = r8     in1 = r8     in2 = r7     conf = 'MUL_MUL_SI

// Data path 4:
CONST       out = r4                               conf = 5
ADD         out = r6     in1 = r6     in2 = r4     conf = 'ADD_ADD_SI

// Data path 3 (continued):
MOV         out = r4     in  = r7
MOV         out = r2     in  = r8
```

Figure 4.14: Assembly instructions from figure 4.12 grouped by which independent data path they belong to.

Edge	Value represented	Output registers
e1	input from r1	r1
e2	input from r2	-
e3	result of: r0 ← r1 ADD r2	-
e4	result of: r3 ← CONST 4	r3
e5	result of: r0 ← r0 SHIFT r3	r0
e6	result of: r4 ← CONST 1	-
e7	input from r5	-
e8	result of: r5 ← r5 ADD r4	r5
e9	result of: r2 ← CONST 3	-
e10	input from r7	-
e11	result of: r7 ← r2 MUL r7	r4, r7
e12	input from r8	-
e13	result of: r8 ← r8 MUL r7	r2, r8
e14	result of: r4 ← CONST 5	-
e15	input from r6	-
e16	result of: r6 ← r6 ADD r4	r6

Table 4.3: All edges from the example data flow graph in figure 4.13, and the corresponding instruction or register.

Since the basic blocks can be arbitrarily large, it may not be possible to fit all the instructions into a single configuration context, in which case scheduling is needed to determine how to best distribute them amongst a number of configuration contexts. This is described in sections 4.8 and 4.9. However, before this is possible, a data model is needed to represent this arbitrarily large data flow graph (DFG). This internal representation can be thought of as the instructions mapped to a core with infinite resources.

The internal representation of the data flow graph is defined in terms of DFG *edges* (or more specifically, the concept of hyper edge [77]). A DFG edge represents a piece of information created by an operation, and passed as input to one or more dependent operations. Specifically, an edge is created for the following:

input registers: An input register brings a value into the basic block from a basic block that was executed previously (possibly the same block, for kernels). This value is represented by an edge.

instructions: An instruction generates a new value (the result) inside the basic block. This value is represented by an edge.

immediates: Immediates that initialise a register directly, must be represented by an edge, in case that register is an output register.

Table 4.3 shows the edges derived from the example assembly in figure 4.12. Additionally, in the data model, the DFG edge stores a list of registers which should hold the value of this edge upon exit from the basic block (i.e. output registers). For each edge, a list of edges that must be calculated before it (*predecessors*), and a list of edges that must be calculated after it (*successors*), are stored. This relationship is shown in figure 4.15.

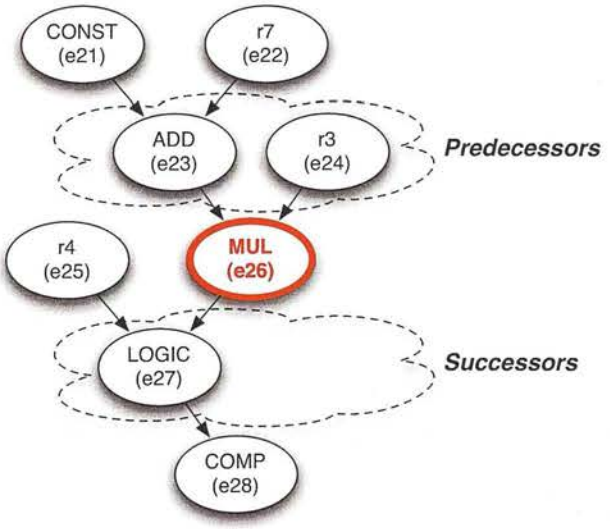


Figure 4.15: Example data path with a particular DFG edge ($e26$) highlighted. Its immediate predecessors ($e23$ and $e24$) and successors ($e27$) are identified.

By abstracting away from the simple register-to-register transfer model captured by the assembly, the inherent parallelism in the data paths becomes immediately apparent. However, certain instructions have side-effects which require special treatment: e.g. memory access operations effectively describe data paths—and therefore connections—that are independent of the registers, which implies a dependency that is not captured in this data model.

To describe these additional connections/dependencies, constraints are stored as part of the data model. A constraint describes a relationship between two DFG edges, and currently can be one of the following:

same step or later: The DFG edge on the left-hand side of the relationship (LHS) must appear in the same step or a step later than the step containing the DFG edge on the right-hand side (RHS).

some step later: The DFG edge on the LHS must be scheduled in a step later in the sequence than the step containing the DFG edge on the RHS.

The main data memory interface of the target architecture performs memory read operations inline—i.e. the result is made available within the same step. However, memory write operations alter the state of the memory only at the end of the step. Therefore, any potentially aliasing memory operations[11] must be placed in different steps. In this case, a *some step later* constraint is placed between the read operation (LHS) and the aliasing write operation (RHS). Similarly, the order of any sequences of writes to potentially aliasing memory locations must be preserved. This is again done using constraints, between those memory write operations. Different memory interfaces with other memory access behaviours can also be modelled in a similar manner.

The resulting data model therefore consists of:

- The list of DFG edges for the basic block.
- The list of constraints involving pairs of these edges.

4.5.1 DFG Analysis Algorithm

Prerequisites: The list of instructions belonging to a live basic block.

Results: An internal representation of the data paths of that basic block, with the inherent parallelism exposed.

The algorithm performs a single pass over the instructions of the basic block, in natural order. A record is maintained of which edge represents the last value stored in each register. This record is called the *register map*.

For each instruction, the operands are scanned. First, each register named in the instruction's inputs is looked for in the register map. Any registers that do not yet appear in the register map represent input registers, so a new edge is created to represent the value read from that input register. For registers that do appear in the register map, the corresponding edge for that register is marked as a predecessor of the edge that will be created for this instruction. Once all inputs have been inspected, an edge is created to represent the output of this instruction, and is recorded in the register map against the register named for the instruction's output in the assembly.

[11] i.e. a write followed by a read from the same address, or what could be the same address.

There is one special case where the behaviour is different: the move (`MOV`) instruction. A move is virtual—there is no corresponding cell in the physical core. A move instruction provides two capabilities that could not be expressed otherwise:

Fan-out: A move instruction with both input and output being named registers, represents a fork in the data flow graph—where the same value is propagated to more than one successor. This is the only way to represent fan-out to multiple output registers. No edge is created for the move instruction in this case; instead the entry in the register map is duplicated for the new register (i.e. both entries refer to the same edge).

Immediates: A move instruction with the input being a literal, represents loading an immediate into a register. An edge is created to represent this immediate (which is treated similar to an input register).

The move instructions constituting the continuation of data path 3 in figure 4.14 are an example of fan-out to output registers.

Where an edge representing an immediate is seen as a predecessor, the immediate is copied to the instruction's input port, and the edge no longer considered to be a predecessor.

When instructions with special side-effects are encountered, appropriate constraints are created. Knowledge of which constraints to create is hard-coded into the tool. Records may need to be kept for certain cases, e.g. for potentially aliasing memory accesses, a record similar to the register map is maintained for the edges that represent memory access operations that could alias to a particular address range.

Once all instructions have been inspected, the final contents of the register map describes the output registers from the basic block. As will be discussed in section 4.7, the assembly has no direct way to state when the value stored in a register becomes unimportant. Therefore, this set of output registers will be a superset of the registers that actually pass information into basic blocks that will be executed later in the program. The live register identification algorithm presented in section 4.7 is used to prune this set of output registers to a subset closer to the true set of live output registers. Note that if any edges representing immediates remain in the register map, then these represent initialising an output register using an immediate, so are preserved in the data model.

4.6 CFG Analysis

The program *control flow graph* (CFG) is a graph describing how control flow may pass between the basic blocks of a program. The nodes are the basic blocks. The graph is directed, and potentially cyclic—due to the presence of loops. This analysis is static—it doesn't execute the program, or use statistics obtained by executing the program. Therefore, no weights are applied to the edges of the graph, meaning that the graph does not contain information about how often or how likely a particular path is followed.

The result of this analysis is a list of all possible jump targets (basic blocks) for each basic block in the program. This information is of vital importance for live register identification (section 4.7), which greatly eases the parallelisation phase (section 4.8); and for the production of global register reallocation information (section 4.12), which can be used by the mapper tool to reduce path lengths and congestion on the final routed configuration contexts.

The control flow graph is complicated by the following:

Recursion: Loops and other forms of recursion introduce cycles in the control flow graph.

Returning from functions: Calling a function is straight-forward—the first basic block of the function is recorded as a jump target for the basic block. However, since each function could be called from more than one place,[12] there will be more than one return point. Furthermore, within the function, there can be more than one basic block that returns. See figure 4.16.

Function pointers: Jumps to program addresses obtained by dereferencing a function pointer must be identified, so that the resulting function call can be correctly taken into account.

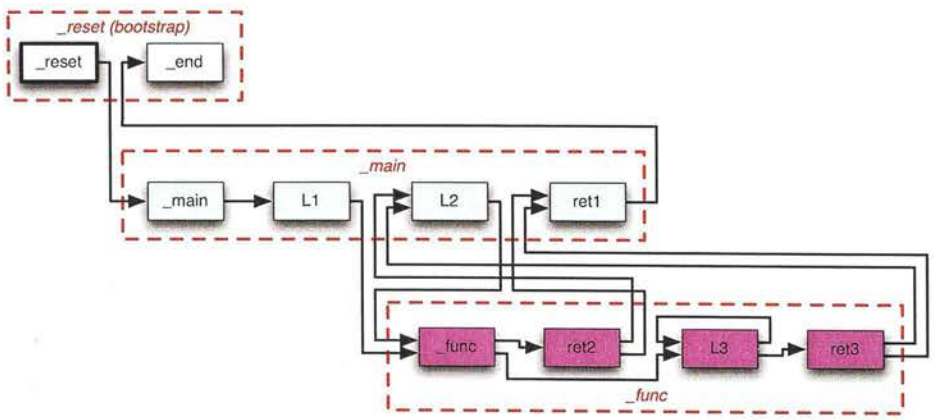


Figure 4.16: Example program control flow graph with three functions (two user-defined, plus bootstrap), shown by the dotted red outline. The basic blocks within each function are shown. The program entry point is shown in bold. The function `_func` is called from two places in `_main`, and contains two blocks that return from the function. This results in multiple jump targets.

[12] otherwise the function should have instead been inlined.

4.6.1 CFG Analysis Algorithm

Prerequisites: Information as to which functions in the program are live (reachable) during execution, which functions have been referred to by address in any live code (section 4.4), and identification of the value supplying the address to the jump instruction (section 4.5).

Results: The list of all possible jump targets (basic blocks) for each basic block in the program.

The basic visitation algorithm is simple: each live function in the program is visited, in arbitrary order. For each live function, the basic blocks belonging to the function are visited recursively in the order that they call each other, beginning with the first basic block in the function (the entry point). The control flow behaviour of each basic block can one of the following:

No jump: The basic block passes control directly to the next basic block in sequence, within the function (e.g. `_main` in figure 4.16). The next basic block is recorded as the sole jump target of the current block, and the next block is visited.

Unconditional jump: The basic block passes control directly to a named basic block within the same function. The named block is recorded as the sole jump target of the current block, and the named block is visited.

Conditional jump: The basic block either passes control to the next basic block in sequence, or passes control to a named basic block (e.g. `_func` and `L3` in figure 4.16). Both targets lie within the same function, and are marked as the only jump targets for the current block. Both of these blocks are then visited, in turn.

Function call (direct): The basic block unconditionally passes control to the first basic block in the named function (e.g. `_reset`, `L1` and `L2` in figure 4.16). This is identified by the fact that the target basic block is not in the same function as the current block. The current block is recorded as a function call, for later reference. The target block is marked as the sole jump target of the current block. However, the called function is not visited at this stage; the next block in sequence from the current block is visited instead, as this will be the return point.

Function call (indirect): The basic block unconditionally passes control to an address obtained by dereferencing a function pointer. The target block could be the first basic block of any function whose address has been referred to in live code (candidates are provided during linking—section 4.4). The current block is recorded as a function call, for later reference. All blocks identified as potential function pointer targets during linking are recorded as the jump targets of the current block. None of these are visited, and instead the next block in sequence from the current block is visited, as this will be the return point.

Return from function: The basic block unconditionally passes control to the address stored in the link register (e.g. `ret1`, `ret2` and `ret3` in figure 4.16). The block can pass control to the next block in sequence from each of the places that the function was called. These are not known at this stage, so the block is recorded for later processing. This marks the end of the control flow within the current function, so the recursive visitation function returns.

Since the control flow graph is typically cyclic, an end point has to be enforced to prevent infinite recursion inside loops. This is done by keeping a record of which basic blocks have already been visited, and returning immediately if the current basic block has already been visited.

Once the visitation function returns from the first basic block in the function, the next live function is visited. Once all live functions have been visited, the potential jump targets can be finalised for each basic block that returns from a function. This is done by consulting the list of which basic blocks called that function.[13] The next block in sequence from each of these callers is recorded as a jump target of the blocks that return from functions.

In reality, if there are more than one basic block that returns from a function, then not all of these will necessarily be reached for each caller. But it is a safe simplification to assume that all return points return to all callers. The side effect of this simplification is more paths of control flow to explore during live register identification, which could lead to additional registers being needlessly marked as live (see section 4.7).

As noted above, to deal with function pointers, a safe assumption is made: *all* functions whose address is referred to in the program, are considered to be jump targets for each basic block that performs an indirect jump. An indirect jump is identified by the jump address being obtained by dereferencing a function pointer. This assumption is unlikely to match reality, and as a result, many registers may be incorrectly seen as being live. This is safe, but again, is potentially inefficient.

Indirect branches can also occur to internal labels, e.g. via a long jump. These should be treated in a similar manner to function pointers, where the address of the basic block corresponding to that internal label will be taken in the code and stored in memory (on the stack).

As a result of the traversal, CFG analysis also yields further dead-code stripping, since any basic block not visited during CFG traversal is, by definition, dead. Easy access to the control flow graph makes a wealth of assembly-level optimisations possible, which are performed prior to parallelisation. These however are beyond the scope of this thesis.

[13] or could have called that function, particularly when function pointers are involved.

4.7 Live Register Identification

Registers are at a premium during the parallelisation phase (section 4.9), and subsequently for pipelining (chapter 5). Without further information, the only safe way to allocate additional registers is to choose those from a pool of registers set aside explicitly for storing temporary values (*scratch registers*). However, setting aside too many registers causes problems elsewhere, such as making the compiler more likely to generate smaller basic blocks, which are harder to parallelise; and setting aside too few increases the chance of register starvation, which again limits the extent to which the basic blocks can be parallelised. This section proposes an algorithm for improving on the estimate of which registers are really live across basic block boundaries, thus making more registers available for other uses.

A limitation of working from the assembly is that the assembly only describes when registers are written to (and hence become live), but not when their stored value becomes unimportant (when they become dead). This is a problem when using the compiler to target the RICA hardware, because the compiler uses registers to pass values between the instructions within a basic block, and many of these values are not used outside of the basic block, which leads to resources being needlessly tied-up. There is no direct way to determine which values are only of relevance internally to the basic block, and which are used to convey information between basic blocks (*output registers*). Also, some registers that are inactive in a basic block may be storing information for use later in the program (*dormant registers*). These too must be identified.

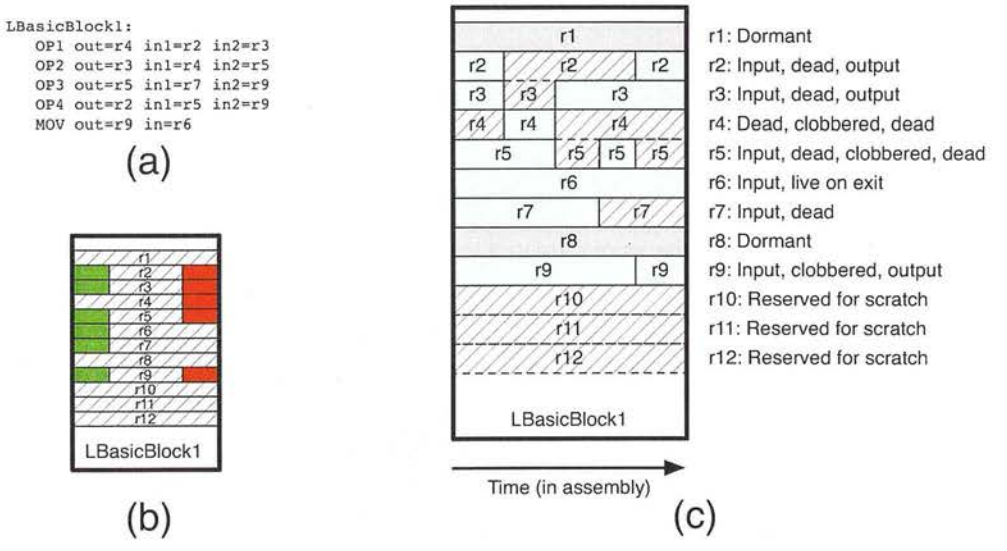


Figure 4.17: (a) Example basic block assembly. (b) Registers read from (input) and written to (output) in the assembly. (c) Corresponding register lifetimes when executing the instructions in the assembly, line by line. The dormant registers and dead output registers cannot be determined by looking at just this basic block alone (and hence are missing from (b)).

Figure 4.17 highlights these concepts, showing how the state of the machine changes during the execution of a simple basic block (figure 4.17(c)). Note that the machine here is what the compiler thinks is the target architecture, which is a simple RISC-like register transfer model; not

the real reconfigurable hardware. A solid fill (cyan or grey) indicates that the register is storing useful information, and a hatched fill indicates that the value stored in the register is no longer needed. In this example, registers $r2$, $r3$, $r5$, $r6$, $r7$, and $r9$ are input registers (shown in green in figure 4.17(b)), which bring data into the basic block for use in computations. During the process of the computations, some of these values become dead (i.e. their value is not needed thereafter), and are overwritten by new values (clobbered—shown in red in figure 4.17(b)). $r2$, $r3$, $r4$, $r5$, and $r9$ are written to in the basic block, but out of these, only the final values of $r2$, $r3$, and $r9$ are needed in later basic blocks (i.e. are output registers). $r1$ and $r8$ are not involved in any operations in the basic block, but are storing information used later on (dormant). Therefore, the registers $r4$, $r5$, $r7$, in addition to the reserved scratch registers $r10$, $r11$, and $r12$, are available for other uses[14]. In this case, this doubles the amount of registers available (i.e. 6 compared to 3). This is quite representative of the situation in general.

The compiler largely knows the register lifetime information, through a combination of the *Application Binary Interface* (ABI) definition, and internal information it stores for the functions and basic blocks as it forms them. However due to internal implementation details, and overly conservative assumptions used in the compiler, it proves difficult to reliably pass information of sufficient quality into the RICA GCC back-end. This is mainly due to what information is discarded before entering the GCC back-end, and how the compiler works on a per-function basis, assuming that every caller saved register it modifies in that function needs to be saved on to the stack, irrespective of whether it was live in any control paths leading to that function. This is primarily a result of the support for incremental compilation, and the ability to call a function from anywhere[15]. An alternative approach was needed.

The key observations are as follows:

- A value stored in a register is by definition live between when it was written and when it is last read from.
- A value stored in a register is by definition dead when it is overwritten (clobbered).

This applies throughout the entire execution of the program. However, since the particular execution control flow path followed during any given run of a program is arbitrary, usually non-deterministic, and potentially cyclic, all possible paths must be considered. Since a single set of configuration contexts is created for each basic block, they have to apply to each possible path in which they could be part of.

As a result of parallelisation (section 4.9), values internal to a basic block that are stored in registers in the assembly, become transferred through wires in the resulting configuration contexts. Therefore, only the state of the registers across boundaries between basic blocks needs to be resolved.

To visualise this, it is useful to draw representations of the registers read from and written to in each basic block, in the form shown in figure 4.17(b). When a register has only a green box, then that indicates that the value is live on entry to that basic block, and is not overwritten. The same value may or may not be live on exit. When a register has only a red box, then that

[14] such as temporary registers, during parallelisation.

[15] e.g. if the function is to be part of a library.

indicates that whatever previous value was clobbered, so by definition is dead on entry to that basic block. The new value may or may not be live on exit. When a register has both a green and a red box, then the previous value is live on entry, but has also been clobbered. The new value may or may not be live on exit. By expressing all the basic blocks in this way, one can manually follow each path of control flow path, watching when a particular register is read from or clobbered. This is essentially what the live register identification algorithm does, and an example can be seen later in that section.

4.7.1 Contribution: Live Register Identification Algorithm

Prerequisites: All possible paths of control flow must be known. This consists of knowledge of all possible jump targets for each basic block. This must include the effect of function pointers—i.e. all locations which they could dereference to.

Results: The set of registers live on exit from each basic block.

The program control flow graph is a directed and potentially cyclic tree. The nodes are basic blocks. The information is initially scattered in pieces amongst the nodes of the control flow graph, coded in the form of which registers provide input to and which registers are overwritten in each basic block[16].

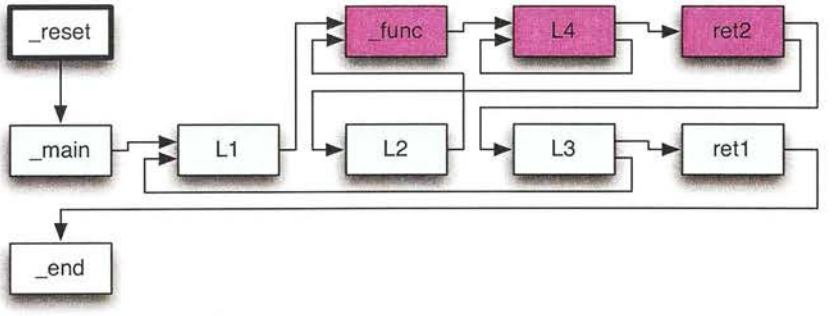


Figure 4.18: Example program control flow graph. The program entry point is shown in bold. The program consists of two functions: `main` (cyan) and `func` (magenta), where `main` calls `func` from two different places. Each function contains a loop.

An example control flow graph is given in figure 4.18. The prominent features of this example control flow are shown in figure 4.19. The example consists of a boot strap (**`reset`**) which calls the function **`main`**. **`main`** contains a loop **`L1`**, **`L2`**, **`L3`** (shown in figure 4.19(d)), and then returns to the bootstrap. This loop in **`main`** calls the function **`func`** twice—once from **`L1`** (figure 4.19(a)), and once from **`L2`** (figure 4.19(b)). Inside **`func`** there is another loop, **`L4`**, shown in figure 4.19(c). This example is complex enough to demonstrate the common issues in identifying live registers: loops, and code that is common to multiple control flow paths (i.e. the **`func`** function being called from more than one place).

[16] which is readily identifiable by inspecting the instructions within that block.

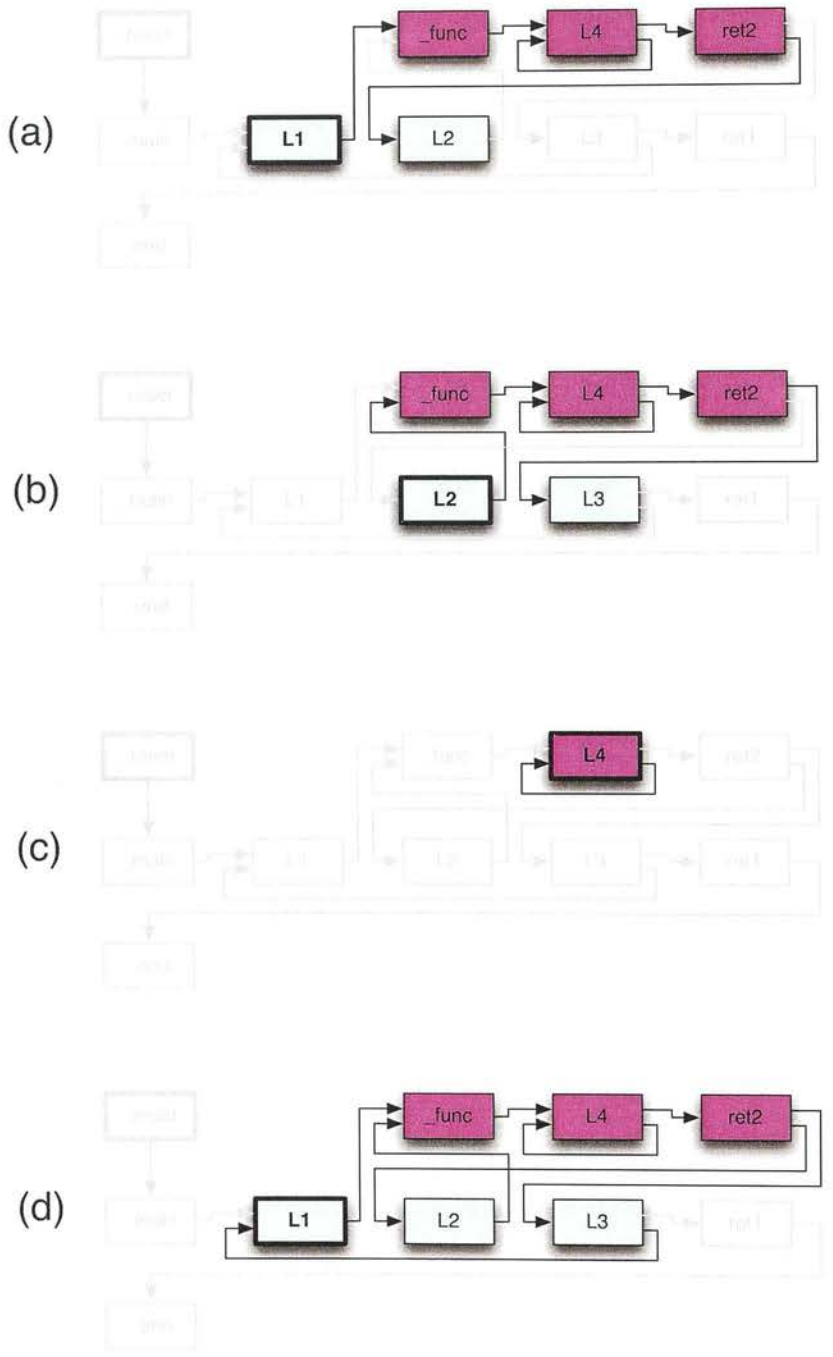


Figure 4.19: Significant features of the example CFG from figure 4.18. (a) First call of the function `func`: from `L1` of `main`, returning to `L2`. (b) Second call of the function `func`: from `L2` of `main`, returning to `L3`. (c) Inner loop: `L4` in `func`. (d) Outer loop: `L1` → call(`func`) → `L2` → call(`func`) → `L3` in `main`.

The proposed algorithm essentially consists of visiting each node in the control flow graph, and 'pinging' each register down the control flow graph stemming from that point, listening for a response as to whether that register has its value used, or whether it has been clobbered. The information gathered at that node is stored, and once complete, is passed back to the previous node. By performing the walk, the complete picture is gradually built up, and the final (complete) information can be seen to be returned from the program entry point.

The walk begins at the program entry point (`_reset` in the example). At each step of the walk, a packet of information is constructed that will be returned to the caller (i.e. the previous step in the walk). This packet of information is the set of registers found to be live on exit from that basic block, according to the branch of the control flow graph stemming from that node. This will be a subset of the registers that are live on exit from the basic block, when considering the entire control flow graph. Once the control flow graph has been fully explored,[17] the complete set of live registers on exit from each basic block will be known. The complete trace of the walk for the example can be found in appendix B.

Basic block	Input registers	Clobbered registers
_reset	-	r1, r2, r9
_main	r1, r2, r9	r1, r2, r5
L1	r5	r3, r4, r9
L2	r5, r11	r3, r4, r9
L3	r5, r11	r6
ret1	r1, r9	r1, r2, r9
_func	r1, r2, r6	r1, r2, r6
L4	r3, r4, r6	r3, r6
ret2	r1, r3, r9	r1, r2, r6, r11
_end	-	-

Table 4.4: Register information for the basic blocks of the example in figure 4.18. The information is read directly from the instructions of each basic block.

The instructions in each of the basic blocks in figure 4.18 yield the information given in table 4.4. Figure 4.20(a) shows this graphically.

Each step of the walk consists of continuing the walk down each branch of the control flow graph stemming from the current node. This is done by visiting each potential jump target for the current basic block. The current basic block is the caller to each of these, and as a result, receives the packets of information resulting from the walk down each of the potential jump targets. Once these packets of information have been accumulated for the current basic block, construction begins for the packet of information that is to be passed to the caller of the current basic block (the LHS in the CFG edges shown in tables B.1 and B.2 in appendix B). This packet contains a list of additional registers that could be read from as a result of going down this branch in the CFG. This list of additional registers consists of the list of input registers to the current basic block, and the list of registers found to be live on exit from the current basic block (so far) minus those that are clobbered in the current basic block.

[17] i.e. where each possible way of visiting each node has been considered.

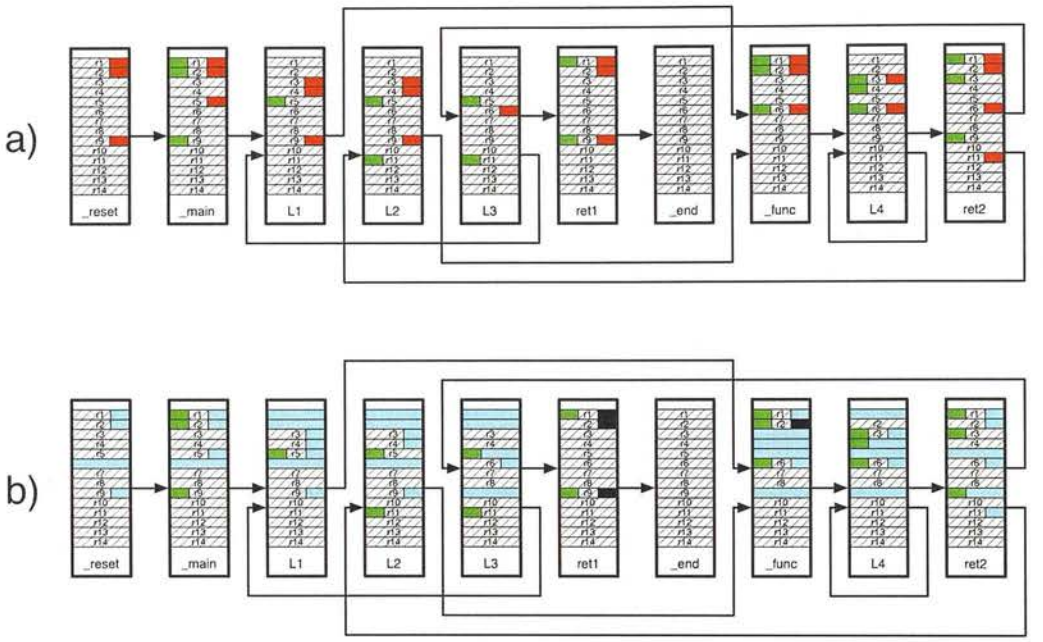


Figure 4.20: The example CFG from figure 4.18 showing (a) input registers (green) and output registers (red) read from the assembly (as shown in table 4.4), and (b) the registers determined by the algorithm to be live (cyan), and the output registers determined to be dead (black).

Basic block	Registers live on exit	Available as temporaries	
		Before	After
_reset	r1, r2, r6, r9	3	10
_main	r1, r2, r5, r6	3	10
L1	r1, r2, r3, r4, r5, r6, r9	3	7
L2	r1, r2, r3, r4, r6, r9	3	8
L3	r1, r2, r5, r6, r9	3	9
ret1	-	3	14
_func	r1, r3, r4, r5, r6, r9	3	8
L4	r1, r3, r4, r5, r6, r9	3	8
ret2	r1, r2, r5, r6, r9, r11	3	8
_end	-	3	14

Table 4.5: Final record of registers live on exit from each basic block in the example in figure 4.18. The effect on the number of registers available for use as temporaries inside each block is shown (where 'Before' can use only the scratch registers). There are 14 registers in total in this example, of which $r12$, $r13$, and $r14$ are reserved as scratch.

This update logic is shown in the rightmost column in the tables, where the new return packet is formed by taking the caller's current list of known live registers, and adding the input registers of the current block, and adding the live registers identified for the current block[18] with clobbered registers filtered out.

To deal with cycles in the control flow graph (i.e. to prevent infinite recursion), a record is kept for each basic block, as to which basic blocks it was visited from during the walk (i.e. which basic blocks control flow passed from). If the same sequence of two basic blocks (i.e. CFG edge) visited is repeated later in the walk, then control does not pass to the potential jump targets of that basic block; instead, the packet that is to be passed to the caller is constructed from the previously obtained information for the current basic block.

At each step of the walk, the live register information is potentially incomplete, depending on how that basic block was visited in the previous walks. The cycle avoidance mechanism particular affects this. To ensure that the information is complete, the entire walk must be repeated until no new information is obtained. This is especially important since the same cycle[19] could appear in more than one branch of the control flow graph. For example, if a function contains a loop, that loop is a cycle that will appear in each branch of the control flow graph where that function has been called. Each time the entire walk is repeated, the record of which basic blocks visit each basic block is cleared. However, the record of live registers on exit from each basic block is *not* cleared.

Table 4.5 shows the final live register information obtained from the CFG walk, collected together. Figure 4.20(b) shows this graphically. It can be seen that all dormant registers have been correctly identified, and a few registers that were written to inside basic blocks were found to be dead on exit from those basic blocks (shown in black in the figure).

As an optimisation note, the implementation of this algorithm makes use of a recursive function. The record of which basic blocks visit each basic block and the record of which registers are live on exit from each basic block are stored outside of the scope of the recursive function. The packets of information are represented by variables (sets) stored on the stack, and are passed by reference to the recursive function. This way, the recursive function updates the value of the variable in the previous stack frame. However, a further optimisation is possible: since the packet of information is a subset of the final complete information for that node, and storage already exists for that complete information for the node, a reference to that storage can be used instead, and passed to the recursive function.

4.7.1.1 Limitations

The algorithm can falsely consider certain registers to be dormant, when in fact the information stored there is never actually needed. Furthermore, some registers are falsely considered to be live before function calls. This particularly affects the program entry point (as can be seen with $r6$ in `_reset` in figure 4.20(b)). This is a further side effect of our use of assembly as the input data model. The compiler considers all registers that it overwrites in the assembly for a function as being live, and therefore pushes their value onto the stack in the function prologue

[18] after visiting all of its jump targets.

[19] i.e. the sequence of basic block names that were visited down one particular branch of the control flow tree.

($r6$ is used in the **func** function, and thus is pushed onto the stack in the basic block **_func**, which causes $r6$ to be seen as live all the way back to the program entry point). However, as mentioned in previous sections, many of the registers that the compiler uses in the assembly become wires, and therefore are not clobbered in the resulting netlist (so don't really need to be stored). The scheduler currently does not treat stack pushes as being special in any way, and as a result, the register whose value is pushed onto the stack is considered live on entry to the first basic block in the function, even if it isn't subsequently overwritten in the function. This live status passes backwards through the control flow graph all the way until that register was last written to.[20] Also, the subsequent popping of the previous value from the stack during the function epilogue similarly results in all pushed registers being seen as live on exit from the last basic block of the function. This can be even worse, since functions can have multiple exit points—all of which will have those registers marked as live on exit.[21]

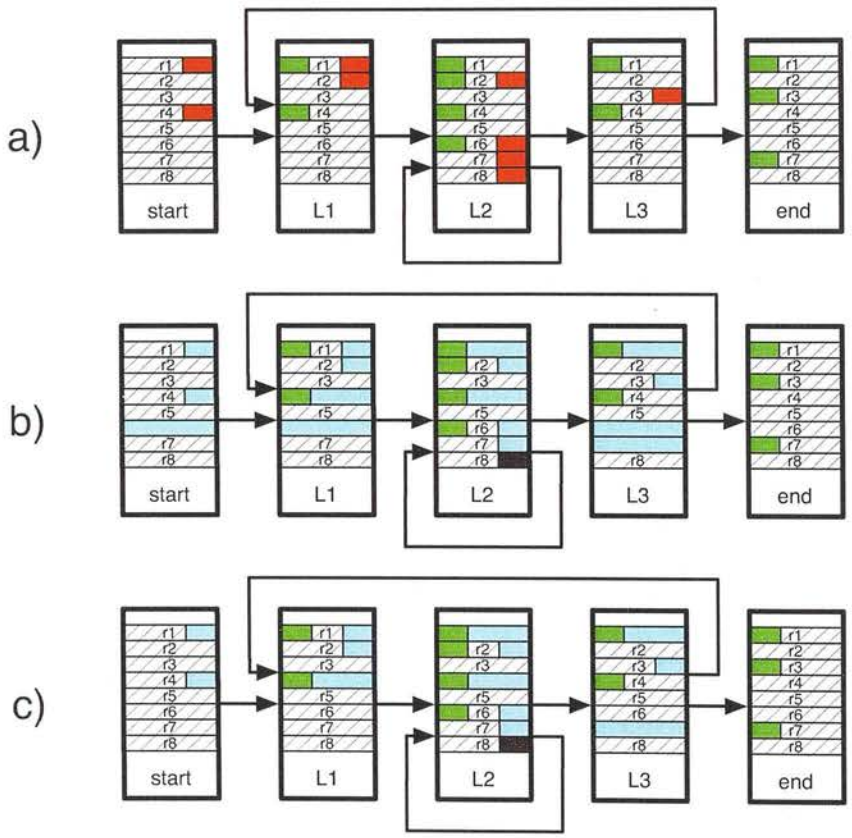


Figure 4.21: Example demonstrating a problem with identifying live registers in nested loops: the algorithm incorrectly considers $r6$ as live in `L1`, `L3`, and `start`, due to the presence of the outer loop. Where: (a) the input and output registers obtained from the assembly, (b) the live registers identified by the algorithm, and (c) the registers that are really live.

[20] this is not the case in the example—$r6$ is actually used to pass information between basic blocks in the **func** function, and so does get clobbered and must be stored.

[21] in the example, this causes $r6$ to be falsely considered live in the block `L2`.

Since the assembly has no way to explicitly define which registers are dead on exit from a basic block, the live register identification algorithm can mark many registers as falsely being live when a block appears inside a loop. e.g. if some registers are really live on exit from a basic block inside the loop, but are not really live on entry to that basic block, then these are still seen as live on entry to that basic block. This is even the case if the registers are clobbered elsewhere in the loop body. An example of this is given in figure 4.21. There are two nested loops: `L2` with $r1$ as the loop counter, and `L1` $\rightarrow$ `L2` $\rightarrow$ `L3` with $r2$ as the loop counter. Inside the inner loop, register $r6$ is used to pass a result to the next iteration of the inner loop. However, due to the presence of the outer loop, $r6$ is falsely marked as live throughout `L1` and `L3` (and indirectly also falsely marked as live in the block `start`). Such false positives do not lead to incorrect program behaviour, but simply prevent certain registers from being used for other purposes.

All of these problems are the result of there being only one record of live registers on exit from each basic block. Since some registers are only live when following certain CFG edges stemming from that basic block, they propagate false positives down the other CFG edges stemming from the same basic block. It may be possible to extend the algorithm to take this into account, by recording a separate live register list for each CFG edge stemming from each basic block.

4.8 Parallelisation

The basic blocks that were determined to be live during CFG analysis (see section 4.6) are each scheduled into one or more steps (configuration contexts). This transformation involves packing the independent data paths (determined through DFG analysis—section 4.5) of that basic block into as few a steps as possible, each with as short a critical path length as possible. This packing is achieved by means of a scheduling algorithm, which is discussed in section 4.9. The second aspect of this task—that of assigning registers to bridge fragments of data paths across the resulting step boundaries—is discussed here.

The resulting sequence of steps should have the same set of input registers and (live) output registers as the original basic block. However, any state changes of registers internal to the basic block need not be honoured. In other words, only the state of registers that are used to transfer values between basic blocks needs to be preserved.

If it is not possible to pack all the instructions from a given basic block into a single step (configuration context), then it will be necessary to infer new registers (termed *temporary registers*) to store the values of any data paths that span the boundary between steps.

In addition to this, registers must be assigned on each step boundary to store temporary values across that boundary. Such values correspond to edges in the data flow graph that straddle across a boundary, when a data path cannot be packed entirely into a single step.

4.8.1 Temporary Register Assignment

A temporary register is assigned to each value that needs to be stored across each step boundary. Even the final value written to output registers in the assembly are stored in temporary registers. The values are only transferred to the output registers in the last step. Similarly, values brought into the basic block via input registers are transferred into temporary registers in the first step. This means that all registers active in the assembly for that basic block, are available for use as temporaries across each internal step boundary, and all values that exist across each step boundary are treated equally. This has the advantage of simplifying the problem, and also increases the number of registers available internally, as will be discussed shortly.

Registers are assigned to store the values represented by edges in the data flow graph, which were identified during DFG analysis (section 4.5). Exactly one register is needed to store each edge that is live over a step boundary. Fan-out (if necessary) is accomplished in the subsequent steps, where multiple operations read from the same register. The calculation of which registers are available for as temporaries is done for each step. The following logic applies:

Input registers: Each input register is represented by an edge in the DFG, and these have to be bridged across step boundaries until the last step where that value is read. The value is transferred into a temporary register on exit from the first step. Note that the value may be live all the way through the basic block, in which case it will be transferred to one or more output registers.

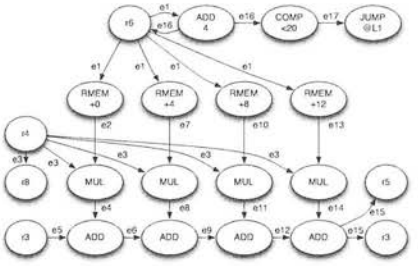
Output registers: These can be used as temporaries in all steps before the final value is written to it. For conceptual simplicity, output registers are only considered to be written to in the last step; a temporary register is used to bring their final value through all the steps since the one where it was calculated.

To prevent needless register to register copying, temporary register allocation gives some edges precedence: temporary registers representing values read from an input register are kept in the same register as the input register, and temporary registers representing values that will be written to output registers are kept in one of the output registers that will receive that value. For all other temporary values, the same register is assigned from one step to the next so long as it is still available.

a)

```
L1:
  RMEM    out= r2    in_addr= r6    in_off= 0     conf= 'READ_SI
  MUL     out= r2    in1= r4        in2= r2       conf= 'MUL_SI
  ADD     out= r3    in1= r3        in2= r2       conf= 'ADD_SI
  RMEM    out= r2    in_addr= r6    in_off= 4     conf= 'READ_SI
  MUL     out= r2    in1= r4        in2= r2       conf= 'MUL_SI
  ADD     out= r3    in1= r3        in2= r2       conf= 'ADD_SI
  RMEM    out= r2    in_addr= r6    in_off= 8     conf= 'READ_SI
  MUL     out= r2    in1= r4        in2= r2       conf= 'MUL_SI
  ADD     out= r3    in1= r3        in2= r2       conf= 'ADD_SI
  RMEM    out= r2    in_addr= r6    in_off= 12    conf= 'READ_SI
  MUL     out= r2    in1= r4        in2= r2       conf= 'MUL_SI
  ADD     out= r3    in1= r3        in2= r2       conf= 'ADD_SI
  MOV     out= r5    in=  r3
  MOV     out= r8    in=  r4
  ADD     out= r6    in1= r6        in2= 4        conf= 'ADD_SI
  COMP    out= r2    in1= r6        in2= 20       conf= 'COMP_LT
  JUMP               in_addr= @L1   in_cond= r2   conf= 'JUMP_NEZ
```

b)



c)

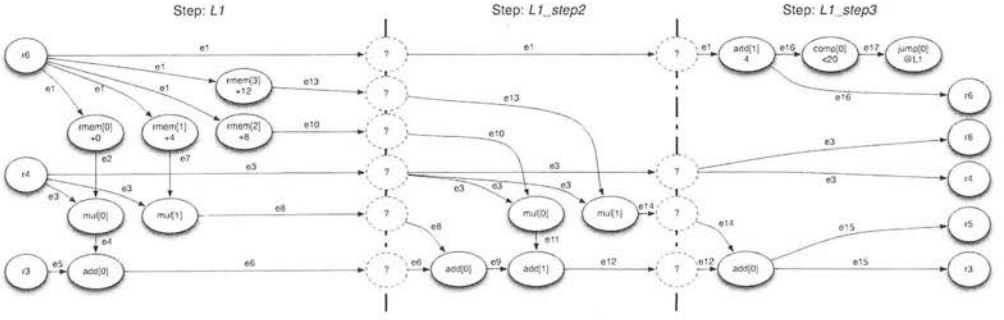


Figure 4.22: Example basic block (loop) showing: (a) the assembly, (b) the data paths extracted from the assembly, and (c) the data paths scheduled into steps, showing the edges that need to be stored (in temporary registers) over each step boundary.

Figure 4.22(a) shows an example basic block (`L1`), and figure 4.22(b) the data flow graph extracted from that assembly. The example is for a 20 term multiply accumulate (MAC), which the compiler has chosen to partly roll into 4 MACs per iteration. The basic block forms a loop, where the final result is obtained in $r5$.[22] The compiler introduces $r3$ as the accumulator, which it initialises to zero before entering the loop. $r6$ is the loop counter, which is incremented (by 4) on each iteration, and is used as the base address for each memory read (RMEM) operation.

To reduce clutter in the diagram, immediates are used for some values (indicated by numbers on the nodes) to feed in constant values. $r4$ supplies the multiplier operand, which is invariant over the iterations of the loop (but may change between successive times the loop is entered).

[22] although this will contain intermediate results until the last iteration.

Its value is duplicated into another register ($r8$), for arbitrary reasons. Since $r4$ is read from in each iteration, its value must be preserved throughout; therefore it is a live output register. The edges are individually numbered—the order of which is determined by where the corresponding instruction appeared in the assembly.

Figure 4.22(c) shows a possible resulting schedule of the same basic block, when resources are limited to 2 ADD cells and 2 MUL cells. This results in 3 steps, where the total critical path has been minimised by distributing the chain of adders across the steps, which minimises the critical path in each step. Also note that the memory read operations are all performed in the first step, since storing their result in registers across the step boundary doesn't increase the critical path of the first step, but reduces the critical path of the second step.

The down-side of this schedule is an increase in the number of edges that are split across step boundaries. Since the assembly only refers to $r2$, $r3$, $r4$, $r5$, $r6$ and $r8$, these are the only registers available as temporaries (total 6). However, in this example there are 3 registers and 4 broken data paths needing to be stored over the first internal step boundary, requiring a total of 7 temporary registers. This would require using a scratch register[23] However, one of the registers being stored is a duplicate of another. By working in terms of edges, and deferring output registers until the last step, registers are freed for each duplicate present. In this case, there is one duplicate ($r8$), so this frees up a register over both internal step boundaries, resulting in there being sufficient registers available as temporaries, without having to rely on scratch.

Note that in general, although this logic reduces the register count, the temporary register count often exceeds the number of active registers in the assembly. This requires the availability of scratch registers, or knowledge about which inactive registers are dead in that block (obtained through live register identification—section 4.7). Similarly, knowledge of which (if any) output registers are dead on exit from the basic block can be used to avoid storing the value across internal step boundaries when not needed.

[23] i.e. a register defined in the ABI as being reserved precisely for this purpose.

4.9 Scheduling Algorithm

In order to achieve the smallest total critical path, all independent data paths in the data flow graph should be executed in parallel. In cases where constraints (such as memory access patterns) make it impossible to perform all in parallel, additional configuration contexts are needed, and the data paths should be placed in as early a context as the constraints allow.

This represents the ideal case for execution speed. In many simple cases, particularly with large cores (with abundant resources), this can be achieved. However, in general, this may not be possible for two reasons:

- There may be insufficient instruction cell resources available in the core to perform all the operations in each configuration context defined by the constraints alone. This requires the insertion of additional step boundaries (configuration contexts).
- Even if sufficient instruction cell resources are available, data paths split across step boundaries imposed by the constraints will require additional registers (referred to as temporary registers) to store the value of each broken edge across the step boundary. If too many edges are split in this manner, there may be insufficient registers available to store all the values. If this happens, the solution is to insert additional step boundaries: forcing smaller broken independent data paths to appear in later steps, instead of the normal policy of as early as possible.

The first issue (resource starvation) is effectively dealt with by means of a ready list. The second issue (register starvation) is more complex, and several strategies have been developed to resolve it (described in section 4.10 on page 95).

4.9.1 Background: List Scheduling

The algorithm is based on list scheduling. List scheduling is an iterative scheduling heuristic, that performs a non-exhaustive search of the solution space. The goal is to pack the given instructions into a series of execution slots, where this packing has the minimum execution time on the target CPU (or some other metric). Successive execution slots are filled by taking entries from the beginning of the ready list.

The ready list is a subset of the instructions that are yet to be scheduled, and contains only those instructions that have no unresolved dependencies. Each entry is checked for constraints being satisfied, and if so, the entry is added to the current slot in the schedule, and removed from the ready list. The ready list is re-evaluated/re-populated each time an instruction is scheduled, since some instructions may now have their dependencies resolved. The algorithm then continues with the next slot. The algorithm finishes once all instructions have been scheduled. The flow chart is given in figure 4.23.

List scheduling algorithms can be applied to multiple issue architectures (such as VLIWs), by flattening the issue slots for each time unit into a single-dimensional array of slots [32]. The logic for calculating whether dependencies and constraints have been met is altered accordingly. This allows the algorithm to do things like populate branch slots [78]—slots after a

branch/jump instruction, which allow certain operations to be performed within the internal latency of the jump. Similarly, the dependency and constraint logic can be modified to take account of pipelined instructions, and can be an effective way of reducing pipeline bubbles [79].

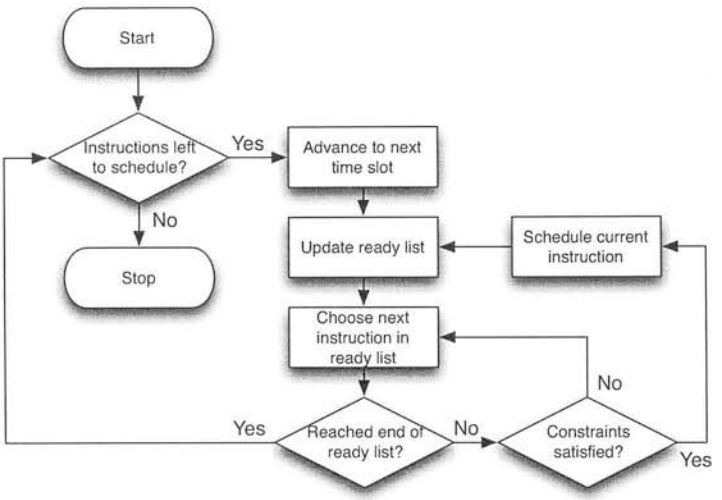


Figure 4.23: Flow chart of generic list scheduling algorithm.

Since the ready list is always inspected in order, its order is very important. List scheduling algorithms differ primarily in how they define this order. The most common sort orders are related to mobility—i.e. the distance that the entry can be moved whilst satisfying its dependencies. Mobility can also be thought of as a measure of idle time or slack. Operations that lie on the critical path have a mobility of zero. Previous work on scheduling on the RICA architecture involved applying mobility-based list scheduling [57].

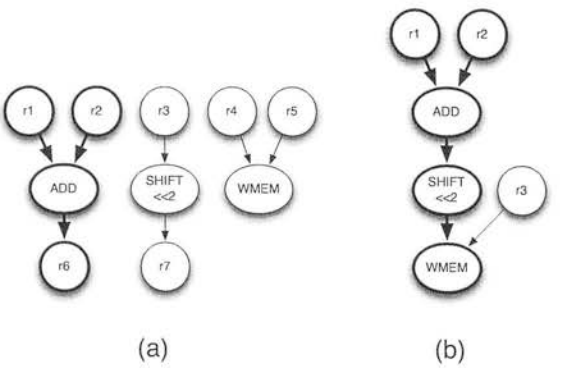


Figure 4.24: Dependent and independent operations. Operations can either be (a) independent of each other, where any inputs and outputs are registers or constants, or (b) dependent on the results of one or more other operations. The latter is called operation chaining. The critical path is shown in bold in each case.

That previous work applied list scheduling to architectures that support operation chaining—i.e. combinatorial data paths. The concept of operation chaining is shown in figure 4.24, where the same operations are connected together in a different manner in the two cases presented[24]. When the operations are independent of one another, these may be executed in parallel, if the hardware supports this. Dependent operations must be serialised on conventional architectures, but can be executed in the same cycle on hardware that supports operation chaining. Operation chaining extends the critical path of the configuration context, which reduces throughput. However, this can be compensated for by pipelining (see chapter 5).

Because the delays of the various functional units and interconnect differ, it isn't accurate to express the configurations in terms of slots of discrete time. A key feature of that work was to extend the concept introduced with multiple-issue architectures, where multiple slots exist for the same unit of time. The execution slot dimension is extended such that a slot is used to represent each available instruction cell resource in the physical array. Then the time dimension is modified such that rather than representing execution clock cycles, each slot represents an entire configuration that should be loaded and executed on the core. Each new time slot represents a new configuration that should be loaded and executed in sequence.

With this strategy, even though the instructions within a configuration look like they are executed in parallel (since they appear in the same 'time' slot), the instructions can be independent (executed in parallel) or dependent (executed in sequence, combinatorially) on other instructions in the same configuration. The original data flow graph is used to determine the connectivity between the instructions within a configuration. The scheduling algorithm need only be aware of the connectivity by the effect it has on the critical path.

There were two mobility-based orderings considered: as soon as possible (ASAP), and as late as possible (ALAP). These refer to where in the ready list instructions with the highest mobility should appear. Instructions with the same mobility appear in the their original (purely sequential) order. ALAP has the effect of giving precedence to instructions that lie on the critical path, but often leads to more sequential behaviour (i.e. lower core utilisation, and more steps). ASAP has the effect of giving precedence to parallel arms of the data flow graph, but often leads to the critical path of the entire DFG not lying on the critical path of each step produced (i.e. additional idle time latency is incurred). Figures 4.25 and 4.26 compares these two approaches using some simple examples.

Clearly, the mobility-based list scheduling approach has its weaknesses. This stems from the fact that the mobility sort order of the ready list places a fixed order of visitation to instructions belonging to different data paths, or to different parallel arms of the same data path. The problem lies when instructions from more than one of these are able to be inserted, which leads to contention. The order in which these arms are added can lead to very different schedules, and in the worst case (i.e. if one of these lies on the critical path) the wrong choice can result in a schedule with a sub-optimal total critical path (i.e. the sum of the critical paths of each configuration in the sequence).

[24] the examples are NOT intended to produce the same result.

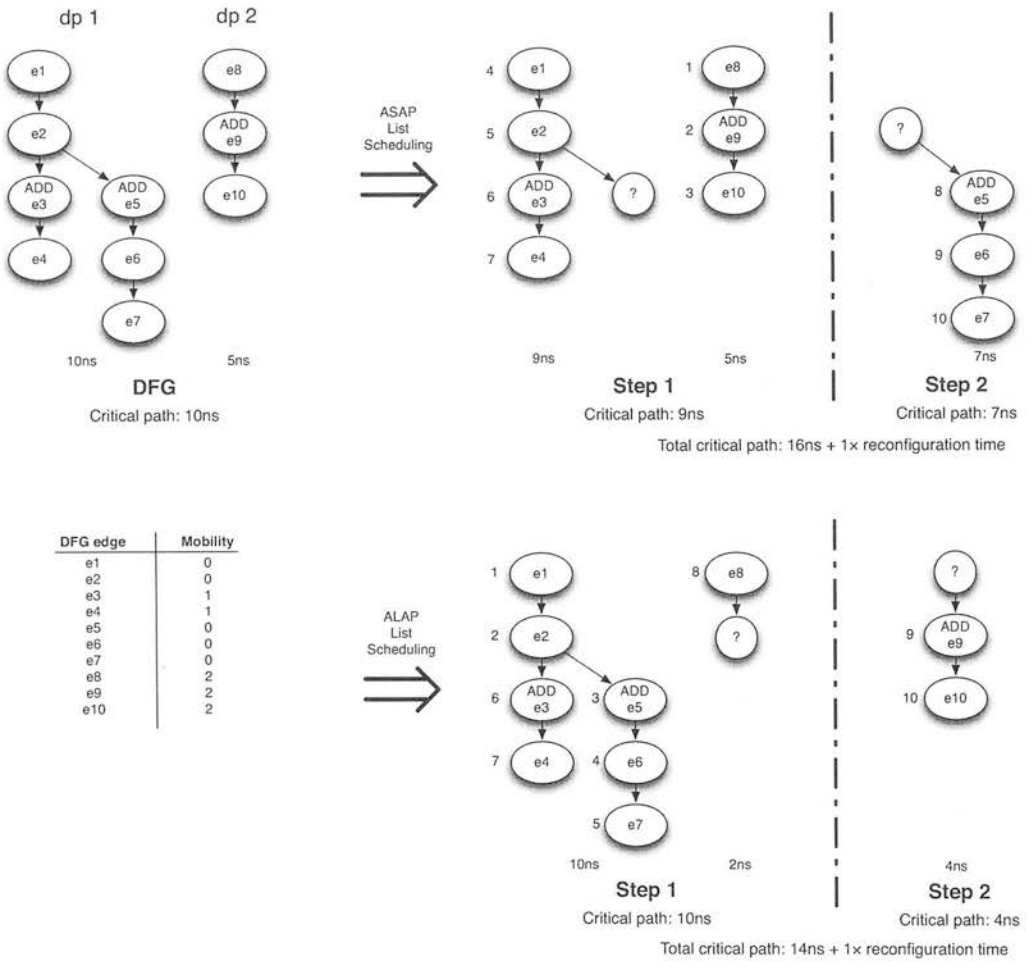


Figure 4.25: Comparison of as soon as possible (ASAP) and as late as possible (ALAP) mobility based list scheduling techniques, with an arbitrary data flow graph. There is contention for the `add` cell resource (there are only two instances of the cell in this example, but three `ADD` operations in the data flow graph). Numbers indicate the order in which the operations were scheduled. In this example, the ALAP scheme prevails.

The mobility sort order enforces a fixed precedence, which can easily cause the wrong arm to be visited (and scheduled) first. Furthermore, when combined with other dependencies later in the data paths, such contention can lead to bubbles (poor cell utilisation) and therefore an excessive number of configurations being generated. This is bad for performance (throughput), and increases the program memory requirements.

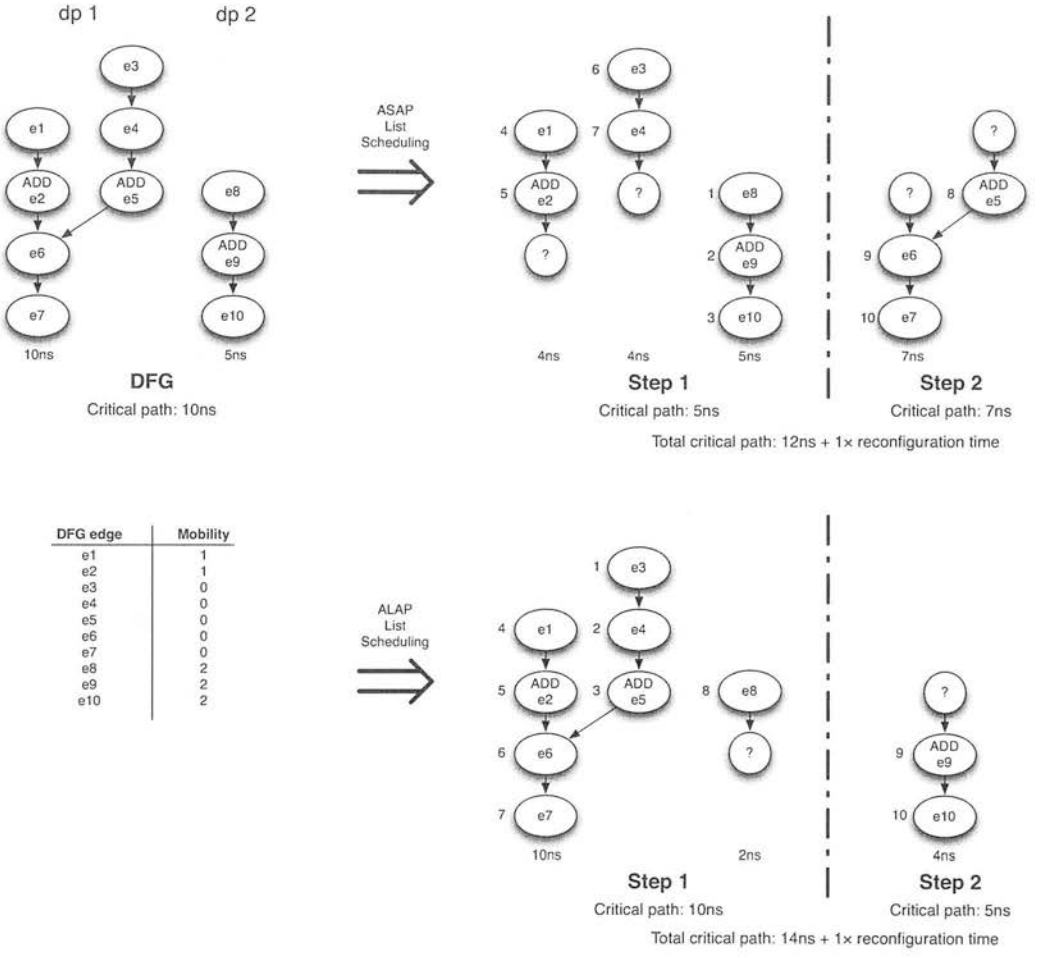


Figure 4.26: Comparison of as soon as possible (ASAP) and as late as possible (ALAP) mobility based list scheduling techniques, with another arbitrary data flow graph. There is contention for the `add` cell resource (there are only two instances of the cell in this example, but three `ADD` operations in the data flow graph). Numbers indicate the order in which the operations were scheduled. In this example, the ASAP scheme prevails.

The goal to minimising the total critical path and/or configuration context count, is to give precedence to the instructions that lie on the critical path in the original data flow graph. These are easy to identify. However, because they are by definition dependent on one another, even if they appear in the ready list before any instructions from other paths, the instructions from the other paths may get scheduled before those on the critical path, due to them becoming ready sooner. Therefore, there is no obvious way to directly encode a fixed visitation order to augment the mobility sort order in the ready list.

4.9.2 Contribution: Tree Follower

The work presented in this section provides a solution to the problem described at the end of section 4.9.1, of how to describe a sort order for the ready list such that the scheduling of instructions not on the critical path do not block instructions on the critical path. In general, no such sort order can be described. The proposed solution is to layer an algorithm on top, to alter the ready list according to the current situation.

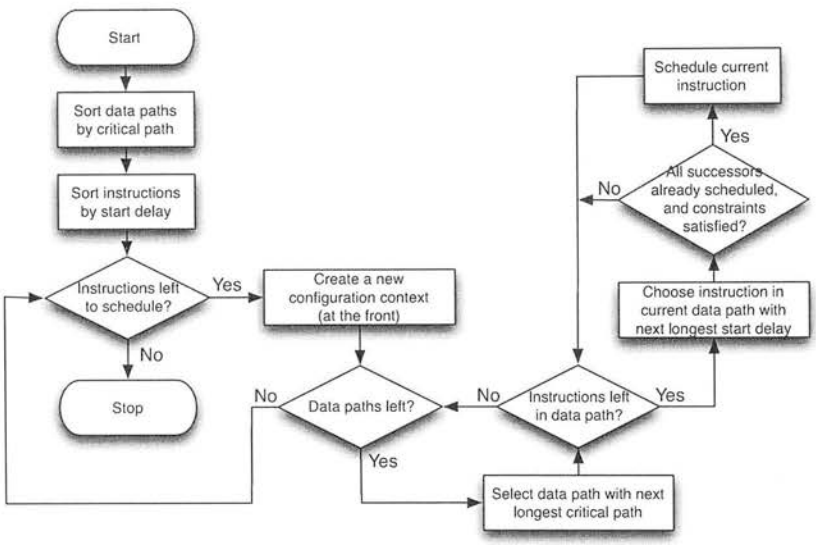


Figure 4.27: Flow chart of the proposed tree follower scheduling algorithm. Note that scheduling is done in reverse order.

The aim of this new layer is to give precedence to the instructions that lie on the current data path (or arm of a data path), only switching to another data path under one of the following conditions:

- once all the instructions on the current data path have been scheduled.
- if no more can be scheduled due to resource starvation (or other constraints).
- if scheduling the current instruction would cause an imbalance compared to other outstanding edges in the ready list.

This reduces the chance of blocking, and should reduce the number of connections split over step boundaries for each data path; thus reducing the register requirement. The algorithm is shown in figure 4.27. Note that no check is performed to check for register starvation when scheduling each edge, as it is likely that starvation may be subsequently avoided by scheduling other edges further up the same data path, depending on the shape of that data path. Instead, a rewind point is saved each time an edge has been scheduled without causing register starvation (see section 4.10.1).

The concept of 'balance' is based on relative position (output delay) in the entire data flow graph. Scheduling is postponed for instructions on the current data path if the difference in output delay between the current instruction and any of the outstanding ones in the ready list exceeds a pre-determined threshold. This avoids needlessly extending the total critical path of all configuration contexts added together.

Essentially, the data paths are added as entries in a second ready list, which is sorted in order of critical path. The data path with the longest critical path is tried first. Whenever a new configuration context (step) is created, the search begins again with the longest remaining data path. Within a data path, whenever there is a choice between arms to descend down, always the one with the longest critical path is considered first[25].

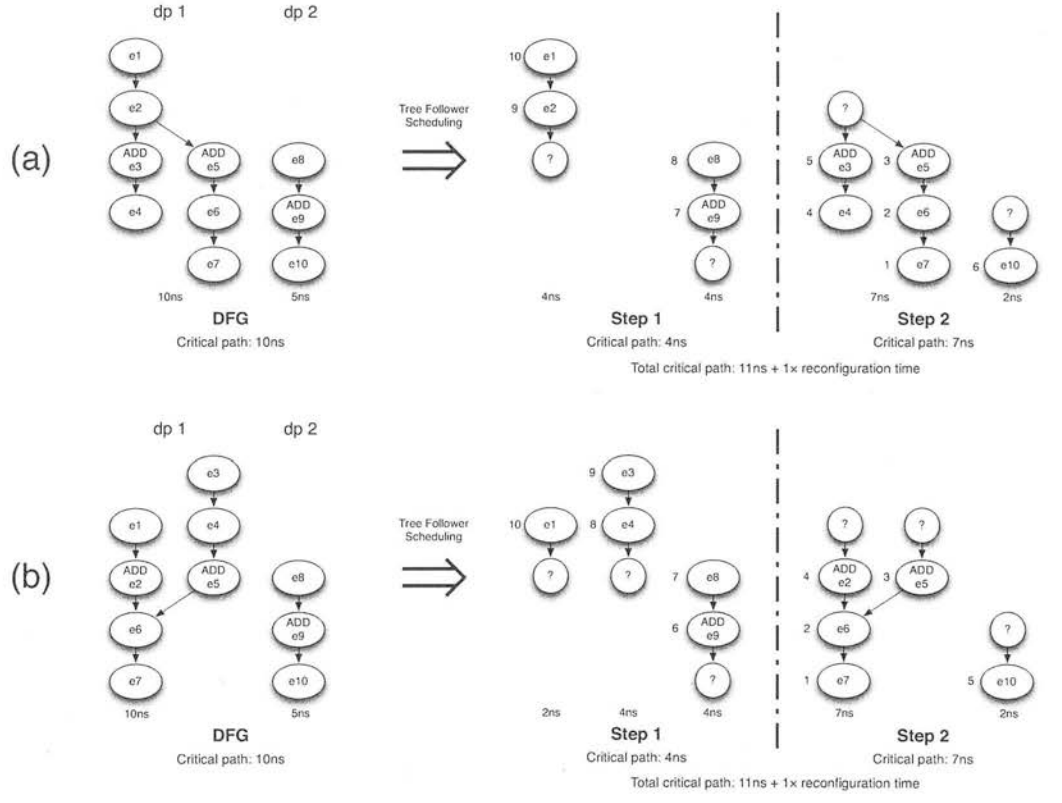


Figure 4.28: The tree follower scheduling algorithm applied to the same examples as before: (a) figure 4.25, and (b) figure 4.26; where only 2 instances of the `add` resource are available. The resulting schedule is more efficient in both cases.

It should be noted that in order to maintain correct program behaviour, the branch/jump instruction (`JUMP`) must be placed in the last step of a basic block. Since the scheduling algorithm is essentially open-ended,[26] it is difficult to ensure this via constraints. One solution would be to delay the insertion of the `JUMP` instruction until an otherwise complete schedule has been created. However, this would lead to a large distance (number of step boundaries) between the

[25] appears earlier in the lower-level ready list.

[26] it keeps creating new steps until there are no instructions left.

JUMP instruction and its predecessors, which consumes registers. An alternative is to simply schedule in reverse, so that the first step created is the last step to be executed in the program sequence. Instructions that have no successors (such as the JUMP instruction) will therefore be scheduled first, thus placing it in the first step generated. Furthermore, the normal operation of the algorithm would ensure that its predecessors be placed nearby, thus reducing—and in most cases eliminating—the need for registers to store their value across step boundaries.

The output of the scheduling algorithm is a data model describing the steps that were generated. Each step lists the DFG edges (see section 4.5) that were scheduled in that step, along with a list of DFG edges whose value must be brought into that step from the next step in sequence.[27] A temporary register is assigned to each of these edges brought in from the next step. Figure 4.29 shows an example of this output data model.

```
Step L1:
{
   Temporary registers bringing values into this step (from previous step):
      None

   Edges in this step:
      e1, e2, e3, e4, e5, e6, e7, e8, e10, e13
}

Step L1_step2:
{
   Temporary registers bringing values into this step (from previous step):
      r6(e1), r8(e3), r2(e6), r3(e8), r4(e10), r5(e13)

   Edges in this step:
      e9, e11, e14
}

Step L1_step3:
{
   Temporary registers bringing values into this step (from previous step):
      r6(e1), r8(e3), r5(e12), r4(e14)

   Edges in this step:
      e15, e16, e17, e18
}
```

Figure 4.29: The step data model produced by the scheduling algorithm for the example in figure 4.22 (on page 85), with an arbitrary assignment of temporary registers to the DFG edges being bridged across each step boundary.

When implementing the algorithm, it is possible to optimise the search order by immediately visiting the predecessors of a freshly scheduled instruction, in descending order of start delay.

The effect of this algorithm is essentially to place the longest data path first (partitioned into steps), then pack the next longest data path around it, and so on. This leads to a very tight packing, and thus a high core utilisation, high throughput, and reduced program memory requirement.

However, despite minimising the number of connections split per data path, visiting each data path often leads to multiple data paths having connections split across step boundaries. Overall, this leads to a higher chance of register starvation. Section 4.10 discusses ways to resolve this.

[27] i.e. since the scheduling is performed in reverse, results are seen to propagate from later steps to earlier steps.

4.10 Register Starvation Avoidance

As discussed in section 4.5, the internal representation of the data flow graph (DFG) of a basic block exposes the parallelism inherent in the data paths described by the instructions. The task of parallelisation (section 4.8) uses this information to determine an optimal packing of these independent data paths so that as many as possible run concurrently. If the complexity exceeds the resources available in the target architecture, then a partial serialisation is chosen using the algorithm described in section 4.9. The scheduler assigns a register to store the value of each broken data path across each step boundary (referred to as temporary registers). If there are insufficient registers available for this purpose, then scheduling fails. This section looks at techniques for avoiding failure.

The particular serialisation chosen by the compiler (expressed in the assembly) ensures that the number of data paths that are broken at any moment in 'time' (from one instruction to the next)—each requiring a register to store their value—never exceeds the number of registers in the target architecture. If it is unable to ensure this, the compiler uses the system data memory[28] to store the excess broken data paths at each moment in time. However, by doing so, the resulting memory access operations require that certain operations are executed in a particular order (i.e. remain serialised), and step boundaries are needed between each state change[29]. This significantly reduces the extent to which parallelisation is possible, and thus the throughput is dramatically affected[30].

Therefore, a trade-off has to be made to ensure that the compiler has enough registers available to avoid using the stack, and yet the scheduler must have enough registers set aside (as scratch) for use as temporary registers. Note that the live register identification algorithm described in section 4.7 goes a long way to circumventing this trade-off, by making available any unused registers that were originally under the control of the compiler. Nevertheless, if the basic block is particularly complex—i.e. requiring many times as many computation resources as there are in the core—there may simply not be enough registers available to allow the optimum partial serialisation to be used. A work-around is needed to allow an alternative partial serialisation to be chosen—one that requires fewer registers. This process is called register starvation avoidance.

Four methods of register starvation avoidance were devised. In descending order of desirability, these are as follows:

Rewind: This essentially pulls out fragments of split data paths from the current step, and forces them to be placed in later steps; possibly at the sacrifice of increasing the overall critical path length, if this data path is part of the critical path. Section 4.10.1.

Shuffle: Discards the previous scheduling result and tries to pack the data paths in a random order, rather than in descending order of critical path length. This is repeated a few times, if necessary. Section 4.10.2.

[28] more specifically, the *stack*.

[29] to allow the intermediate results to pass into and out of the external memory.

[30] particularly when taking into account the memory bandwidth.

Basic block splitting: Splits the basic block in half (or close to half, if 'wires' become split), and schedules each fragment separately. This improves the chance of achieving a valid schedule, as it reduces the number of independent data paths available for scheduling at once. Section 4.10.3.

Serialisation: This is similar to basic block splitting, but even more pervasive. Uses the fact that the serialisation chosen by the compiler is itself a valid (but extremely inefficient) schedule. Works through the instructions in the order that they appear in the assembly, using the registers named in the instructions. Some of these registers become wires, if the instruction can be packed into the same step as the instruction that preceded it. Section 4.10.4.

Whenever register starvation is encountered, the scheduler tries these in the order listed above. The first strategies are less likely to compromise the performance of the resulting code, but are also less likely to be able to resolve the conflict. The need for flat scheduling has yet to be encountered. As a result, flat scheduling has not been implemented. The sections that follow describe these techniques in more detail.

4.10.1 Rewind

If register starvation occurs in the current step during scheduling, this approach goes back to the last valid state (without starvation), and creates a step boundary there. This usually has the effect of pulling the last few (incomplete) independent data paths from the current step, and placing them into a new step. Data paths incompletely scheduled in a step require temporary registers to bridge the step boundary. Therefore, by reducing the number of split data paths, this has the effect of decreasing the number of temporary registers needed. However, this is potentially at the sacrifice of increasing the overall critical path length, if the data path(s) in question are part of the critical path.

The example in figure 4.30(a) shows the data flow graph of a basic block containing a single data path. The example assumes that the core has 4 cells (`add` cells) that support the `ADD` operation. As a result, the entire data path cannot be mapped to a single step. It also assumes that there are no scratch registers available to store temporaries, so just the input registers (5 in total) are available. The input register $r4$ is live on exit. The tree walking scheduling algorithm (described in section 4.9.2) would begin by scheduling $e19$, since it has the longest combinatorial start delay, and then would descent down the branch containing $e8$, scheduling into the current step all the DFG edges in that branch, except those representing input registers. i.e. edges $e1$, $e3$, $e4$, $e5$, $e6$ and $e8$ would be scheduled.

Rewind points are saved each time there is at most five broken edges that would need to be stored. For instance, a rewind point would be saved after having scheduled edges $e19$, $e8$, $e6$, $e5$, $e4$, $e3$ and $e1$, since only edges $e2$, $e7$, $e9$, and $e18$ (4 in total) would need to be stored. It would then continue to the next predecessor of $e19$, i.e. $e18$, which again would be scheduled in the current step—but no rewind point would be saved, since $e2$, $e7$, $e9$, $e16$ and $e17$ (6 in total) would need to be stored. This condition is shown in figure 4.30(b), where $e16$ is being considered next.

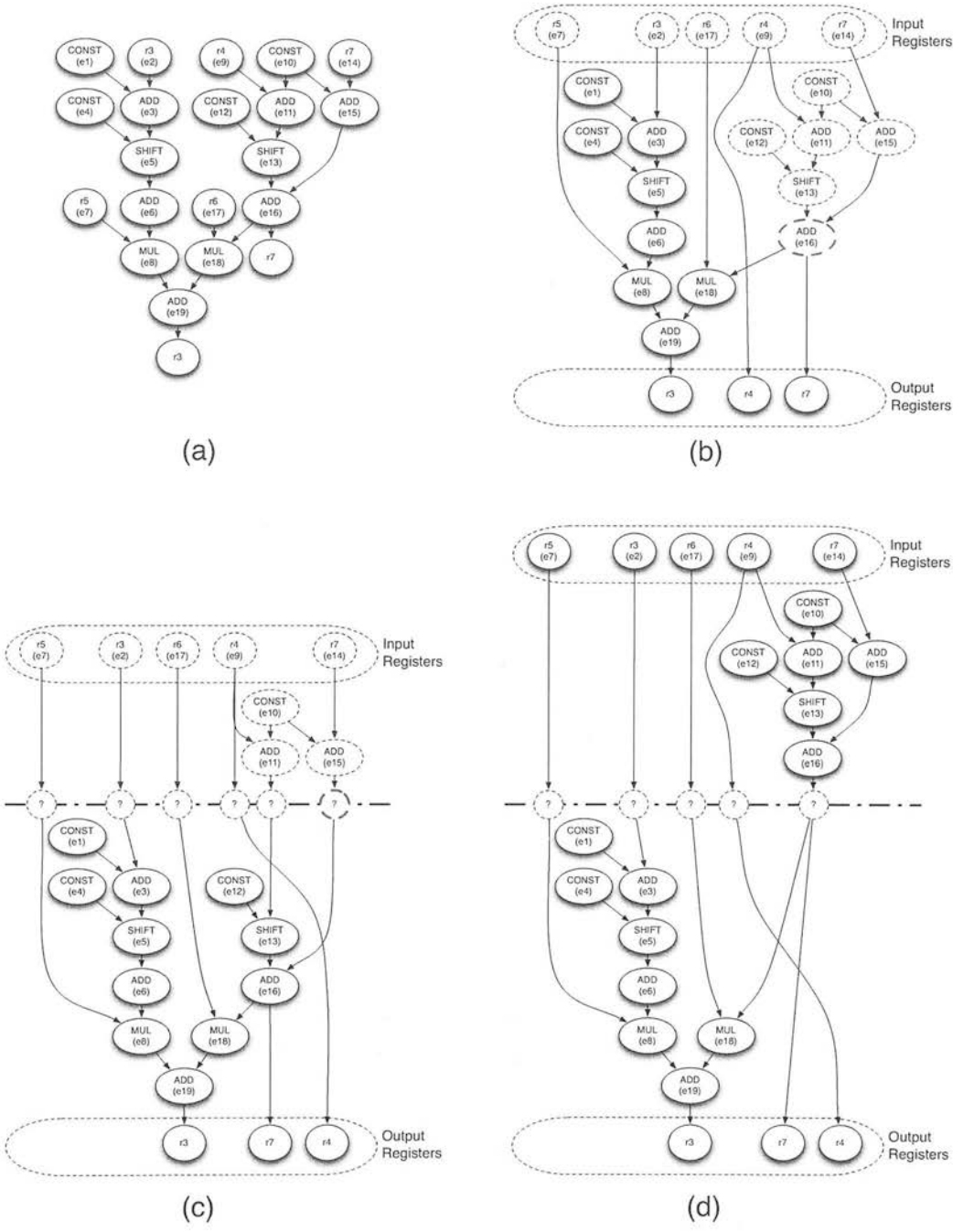


Figure 4.30: (a) Example basic block DFG consisting of one data path. The core has insufficient ADD cells to schedule the whole block into a single step. (b) Last rewind point (before scheduling e16), no register starvation. (c) The resulting schedule if e16 was scheduled in the current step, requiring six temporary registers, leading to starvation (missing register shown in red). (d) The resulting schedule after rewind—i.e. with e16 delayed until the next step, leading to a valid schedule.

Scheduling $e16$ in the current step exhausts the available `add` cells, making it impossible for $e11$ or $e15$ to be scheduled in the current step. The scheduling algorithm would continue to schedule $e13$ and $e12$, resulting in register starvation, as shown in figure 4.30(c). If the rewind register starvation avoidance scheme were then invoked, the situation shown in figure 4.30(b) would be restored, and a new step boundary created. Scheduling would then continue in the next step, and a valid schedule would be achieved, as shown in figure 4.30(d).

This approach has the limitation that although it may be possible to make the current step valid (i.e. free from starvation), there is no account taken of how this will affect subsequent steps. This is the normal mechanism of failure for this approach—rewinding allows a data path to be partially contained in the current step, but the registers incurred to store the broken edges often lead to starvation in later steps; whereas if the data path had been completely postponed until a later step, starvation would have been avoided.

4.10.2 Shuffle

This is based on a random re-ordering of the root nodes. The previous schedule is discarded, and scheduling starts over from the beginning again. This time the independent data paths are considered in a random order, instead of in descending order of critical path length. This has the effect of increasing the chance of smaller data paths being scheduled first. Smaller data paths are less likely to require splitting, and thus are likely to consume fewer registers. At the same time, this has the effect of causing fewer of the longer data paths from appearing in parallel (i.e. serialises them). This potentially increases the total critical path length of the resulting schedule (all steps), which compromises throughput. Rewind is always used in addition to this technique, to prevent too many of the smaller data paths from being packed into earlier steps.

Figure 4.31(a) shows an example basic block consisting of three independent data paths. The core is limited to one `mul` cell (`MUL` instruction), one `add` cell (`ADD` instruction), and two `const` cells (`CONST` instruction). There are no scratch registers available for storing additional temporaries. As a result, only the input registers ($r2$, $r3$, $r4$, and $r6$—a total of four) are available for storing temporaries. The normal root node order would cause edges from data path 2 to be visited first (since the critical path is longest), then data path 3, then data path 1. So, scheduling would begin with $e12$ (which has the longest start delay), then proceeds up that data path to schedule $e11$, $e10$, and $e9$. It can't schedule $e8$, since the `mul` resource has already been used. This exhausts data path 2 for this step, so moves onto $e19$ (in data path 3), which is of the next longest critical path. $e19$, $e18$, and $e17$ can all be scheduled, but $e16$ can't due to starvation of the `add` resource. $e3$ (of data path 1) is visited next, but cannot be scheduled, for the same reason. No more edges can be scheduled, so a new step is begun. There are 4 temporaries needing stored ($e3$, $e8$, $e10$, and $e16$), which is within the availability. The situation is shown in figure 4.31(b).

Scheduling of the next step begins by going back to data path 2, so schedules $e8$, $e6$, $e7$, and $e4$. This exhausts data path 2, so data path 3 is visited. $e16$ and $e15$ are scheduled, but $e13$ can't due to starvation of the `const` resource. Data path 1 is then visited, but $e3$ can't be scheduled due to starvation of the `add` resource. No more edges can be scheduled in the current step, so a step boundary is added. Doing so here requires temporary registers to store $e3$, $e5$, $e10$, $e13$, and $e14$ (a total of five), leading to register starvation, as can be seen in figure 4.31(c). Unfortunately,

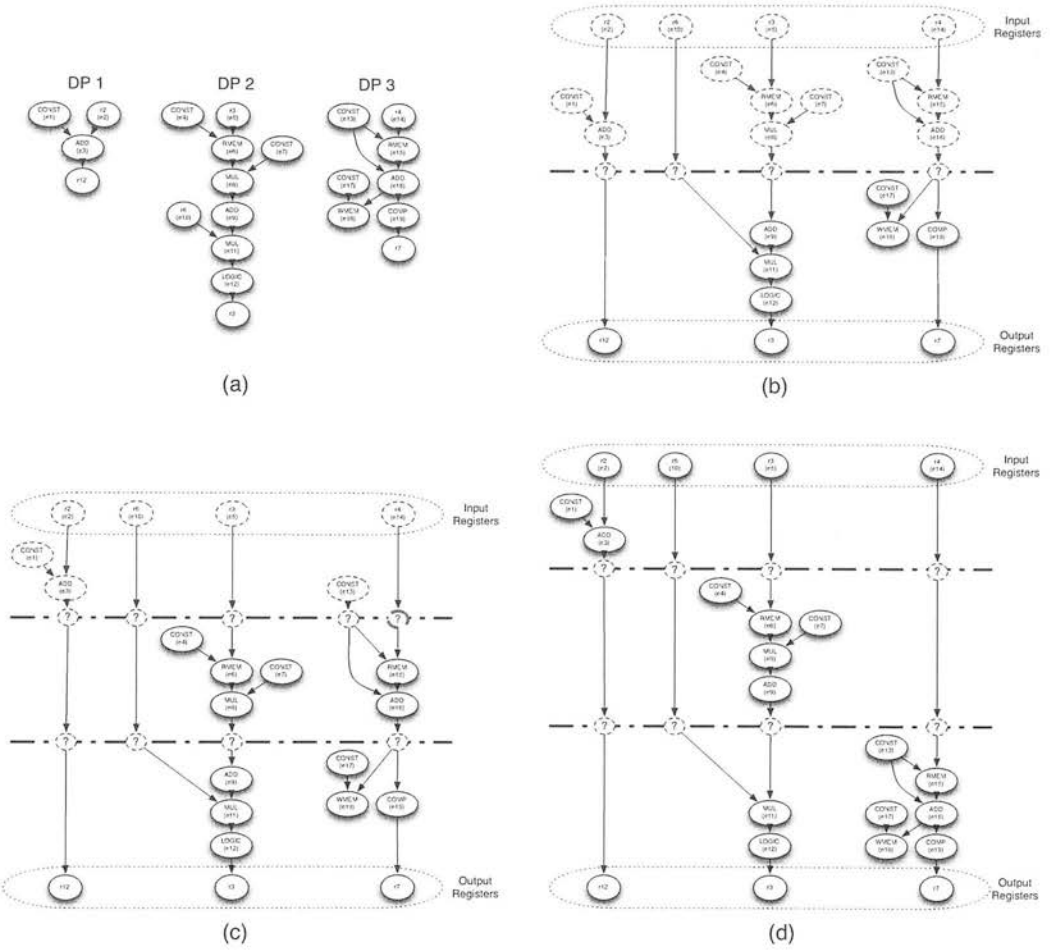


Figure 4.31: Example basic block DFG demonstrating the need for shuffling of the data path search order as a means to avoid register starvation. (a) The data paths and DFG edges, (b) The first step created using natural (depth first) search order (yet to be scheduled edges are shown with a dotted outline). Register starvation has not yet occurred. (c) The second step created, leading to register starvation (missing register shown in red), with no valid rewind point. (d) A valid schedule resulting from visiting the data paths in a shuffled order.

there is no rewind point to go back to, since too many temporaries were needed following the insertion of each edge in this second step. i.e. rewinding would completely unwind that step, leading to exactly the same choices to be made on the next attempt.

This failure can be avoided by discarding that schedule, and beginning again with a shuffled root edge list. This has the effect of visiting the data paths in a different order. The result is shown in figure 4.31(d). Here, data path 3 is visited first, causing it to be scheduled completely in the first step. Data path 2 is then visited next, which can be scheduled up to $e11$, with $e9$ having to be in the next step due to to starvation of the `add` resource. Similarly, $e3$ (of data path 1) cannot be scheduled. A step boundary is created, requiring 4 edges to be stored in temporary

registers, which is within the availability. Scheduling of the next step begins with $e9$ of data path 2, since data path 3 has been exhausted. The remainder of data path 2 can be scheduled without problem. $e3$ of data path 1 cannot be scheduled, due to starvation of the `add` resource, so another step boundary is created. This requires 4 temporaries, which again is within the availability. The remaining edges ($e1$ and $e3$) can be scheduled, and the input registers placed.

4.10.3 Basic Block Splitting

In this approach, the schedule and data flow graph are discarded, and the basic block is modified at the instruction level to split it into two smaller pieces. The purpose of this is to reduce the number of independent data paths that need to be scheduled together. It also leads to large data paths being broken into smaller pieces. A split point is chosen near to the middle of the basic block[31]. The split point may have to be adjusted in order to avoid breaking any *wires*[32], otherwise some data paths will be broken, which would likely change the behaviour of the program.

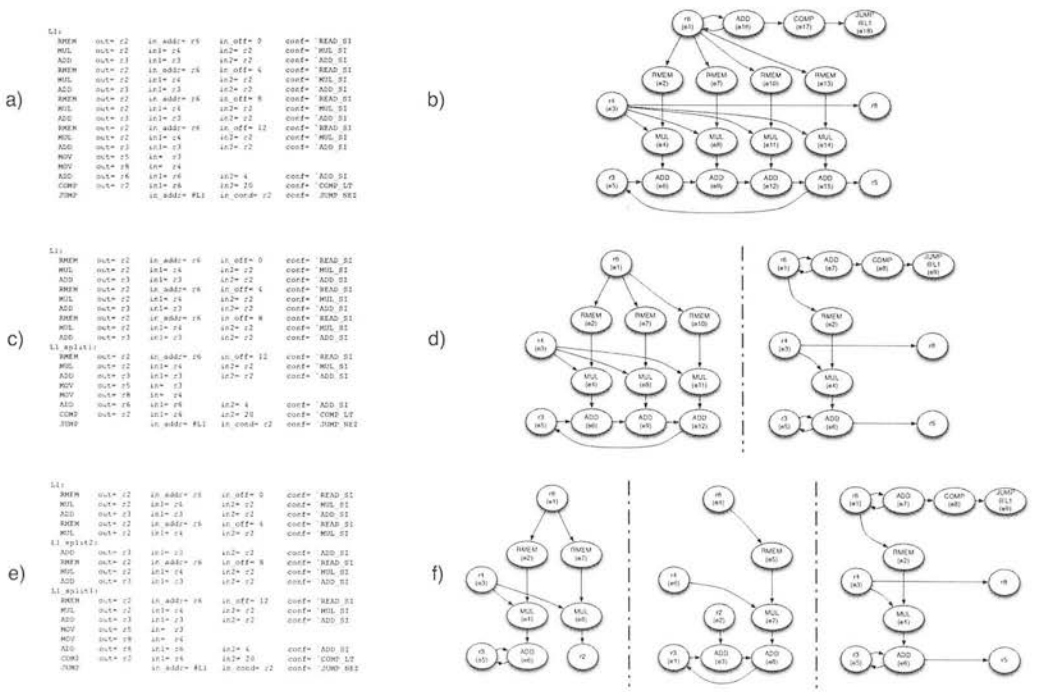


Figure 4.32: Example of basic block splitting. (a) Original basic block assembly. (b) Original data flow graph. (c) Assembly after splitting into two blocks. (d) Resulting data flow graphs of the two new basic blocks. (e) Assembly after another split. (f) Resulting data flow graphs of the three new basic blocks.

[31] i.e. where roughly the same number of instructions appear in each new fragment.

[32] connections that require both end points to be in the same step.

A simple example is shown in figure 4.32. The basic blocks created by the split are then treated separately, as if they had appeared in the original program. This process of splitting can happen any number of times, making the blocks increasingly smaller, to the limit of approaching similar behaviour to the *serialisation* approach, described in section 4.10.4.

4.10.4 Serialisation

This approach involves creating a new schedule by working through the instructions in the order that they appear in the assembly, using the registers named in the instructions. If the instruction can be packed into the same step as the instruction that preceded it, then the register named for that connection can become a wire. This approach makes direct use of the fact that the compiler has already come up with a valid assignment of 'temporary' registers for the whole basic block, using the pool of registers under its control. A step boundary is placed under the following conditions:

- There are insufficient resources to place the current instruction into the current step.
- The constraints require a step boundary at this point.
- The register that is to store the result of the current instruction is already going to be written to by an instruction in the current step.

The last is an interesting point: after scheduling more instructions, some of these will become wires again, in which case this instruction doesn't conflict[33], but this is not known until after scheduling more instructions.

[33] i.e. two instructions are not trying to write a value to the same register in the same step.

4.11 Resource Configuration

The final stage of creating an abstract netlist for the target architecture is to convert the internal data model into a form that is compatible with the netlist syntax. The netlist syntax describes the configuration contexts (*steps*) in terms of active instruction cells, their configuration, and their connectivity. The internal data model chosen for the tasks of parallelisation (section 4.8) and pipelining (chapter 5) operate on data flow graph (DFG) edges (section 4.5). These edges are with reference to the data flow graph of the original basic block. The scheduling algorithm will have packed these DFG edges into one or more steps. So, the data flow graph of these individual steps must then be reconstructed, and the connectivity extracted. This section shows how this can be done.

A DFG edge can either represent reading from a register, reading from the output of an instruction cell, or reading an immediate value that should be stored in a register. Furthermore, the data model for the parallelisation phase describes registers that have been assigned to store temporary values across step boundaries, and pipelining describes registers that have been assigned as pipeline stage registers. All of these must be taken into account when reconstructing the individual step data flow graphs.

Figure 4.33 continues the parallelisation example from figure 4.22. Figure 4.33(a) shows the internal representation obtained during parallelisation. This corresponds to figure 4.22(c), after temporary registers have been assigned on each step boundary. Note that since the schedule in constructed in reverse, the temporary registers are seen to be bringing values of stored edges into the step. Temporary registers were assigned first from the group of active registers in the basic block ($r3$, $r4$, $r5$, $r6$, $r8$), and then from a pool of scratch registers ($r10$, $r11$, $r12$, $r13$). Figure 4.33(b) shows the individual step data flow graphs that should be derived from the internal representation, and figure 4.33(c) shows the same information expressed in the RICA netlist syntax. Temporary register assignment has minimised the number of register-to-register copies needed, so some registers storing temporary values across the step boundaries do not get written to, since they continue to store the same value over each boundary, where possible (which is in all cases in this example).

The process can also choose to re-allocate certain cell instances—i.e. choose to swap certain instances of the same type—where the order matters. The only situation encountered thus far that requires this, is the data memory combinatorial read access cells (RMEM).

Step: L1

Temporary registers bringing edges into step:	*N/A*
Edges in this step:	*e1, e2, e3, e4, e5, e6, e7, e8, e10, e13*

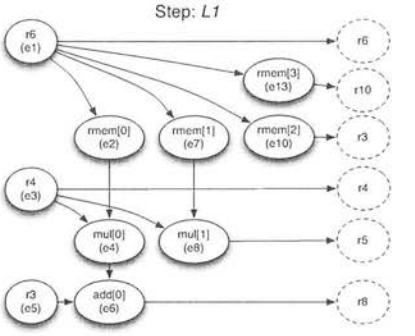
Step: L1_step2

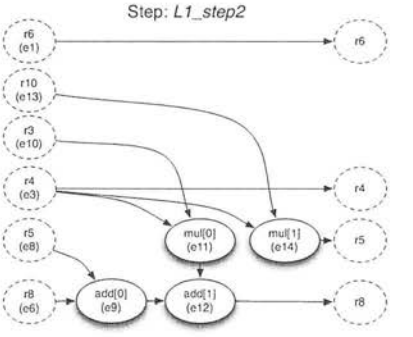
Temporary registers bringing edges into step:	*r6(e1), r4(e3), r8(e6), r5(e8), r3(e10), r10(e13)*
Edges in this step:	*e9, e11, e12, e14*

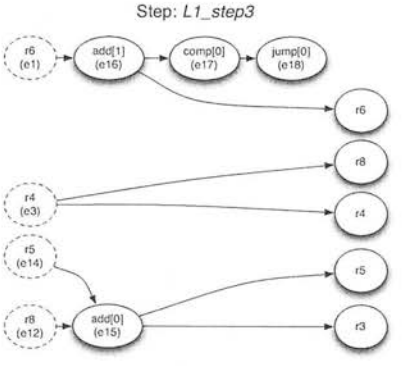
Step: L1_step3

Temporary registers bringing edges into step:	*r6(e1), r4(e3), r8(e12), r5(e14)*
Edges in this step:	*e15, e16, e17*

(a)








```
sequence
{
   step[%L1]
   {
      reg[6];  // e1(in), e1(temp-out)
      reg[4];  // e3(in), e3(temp-out)
      rmem[0] { in_addr = reg[6].out; conf="RMEM_READ_SI; }  // e2
      rmem[1] { in_addr = reg[6].out; conf="RMEM_READ_SI; }  // e7
      rmem[2] { in_addr = reg[6].out; conf="RMEM_READ_SI; }  // e10
      rmem[3] { in_addr = reg[6].out; conf="RMEM_READ_SI; }  // e13
      mul[0] { in1 = reg[4].out; in2 = rmem[0].out; conf="MUL_MUL; }  // e4
      mul[1] { in1 = reg[4].out; in2 = rmem[1].out; conf="MUL_MUL; }  // e8
      add[0] { in1 = reg[3].out; in2 = mul[0].out; conf="ADD_ADD; }   // e6
      reg[8] { in = add[0].out; }   // e6(temp-out)
      reg[5] { in = mul[1].out; }   // e8(temp-out)
      reg[3] { in = rmem[2].out; }  // e5(in), e10(temp-out)
      reg[10]{ in = rmem[3].out; }  // e13(temp-out)
   }
   step[%L1_step2]
   {
      reg[3];   // e10(temp-in)
      reg[4];   // e3(temp-in), e3(temp-out)
      reg[6];   // e1(temp-in), e1(temp-out)
      reg[10];  // e13(temp-in)
      add[0] { in1 = reg[8].out; in2 = reg[5].out; conf="ADD_ADD; }   // e9
      mul[0] { in1 = reg[4].out; in2 = reg[3].out; conf="MUL_MUL; }   // e11
      mul[1] { in1 = reg[4].out; in2 = reg[10].out; conf="MUL_MUL; }  // e14
      add[1] { in1 = add[0].out; in2 = mul[0].out; conf="ADD_ADD; }   // e12
      reg[8] { in = add[1].out; }  // e6(temp-in), e12(temp-out)
      reg[5] { in = mul[1].out; }  // e8(temp-in), e14(temp-out)
   }
   step[%L1_step3]
   {
      reg[4];  // e3(temp-in), e3(out)
      add[0] { in1 = reg[8].out; in2 = reg[5].out; conf="ADD_ADD; }  // e15
      add[1] { in1 = reg[6].out; in2 = 4; conf="ADD_ADD; }           // e16
      comp[0] { in1 = add[1].out; in2 = 20; conf="COMP_LT; }         // e17
      jump[0] { in_addr = @L1; cond = comp[0].out; conf="JUMP_IF_NEZ; }  // e18
      reg[3] { in = add[0].out; }  // e15(out)
      reg[5] { in = add[0].out; }  // e14(temp-in), e15(out)
      reg[6] { in = add[1].out; }  // e1(temp-in), e16(out)
      reg[8] { in = reg[4].out; }  // e12(temp-in), e3(out)
   }
}
```

(c)

(b)

Figure 4.33: Example from figure 4.22 (on page 85) converted to steps. (a) the internal data model from parallelisation, (b) the individual step data flow graphs, (c) the netlist generated.

4.11.1 Background: RMEM Cascading

Each RMEM (read data memory) cell (cell name `rmem`, instruction `RMEM`) samples its address input a programmable number of clock cycles after the beginning of the step, to allow the address value to settle. The memory fetch then begins, returning the value at the output of the cell, within the same step. Due to the presence of arbitration logic, conflicting memory accesses impose an additional (dynamic) delay, as the contending accesses are queued (serialised). The step duration counter is frozen until the queues are empty, to allow the life time of the step to be extended so that all outputs settle before the next step begins.[34]

Dependencies between different memory reads within the same step are expressed via the cell's configuration, and this requires special treatment. This section describes the considerations involved.

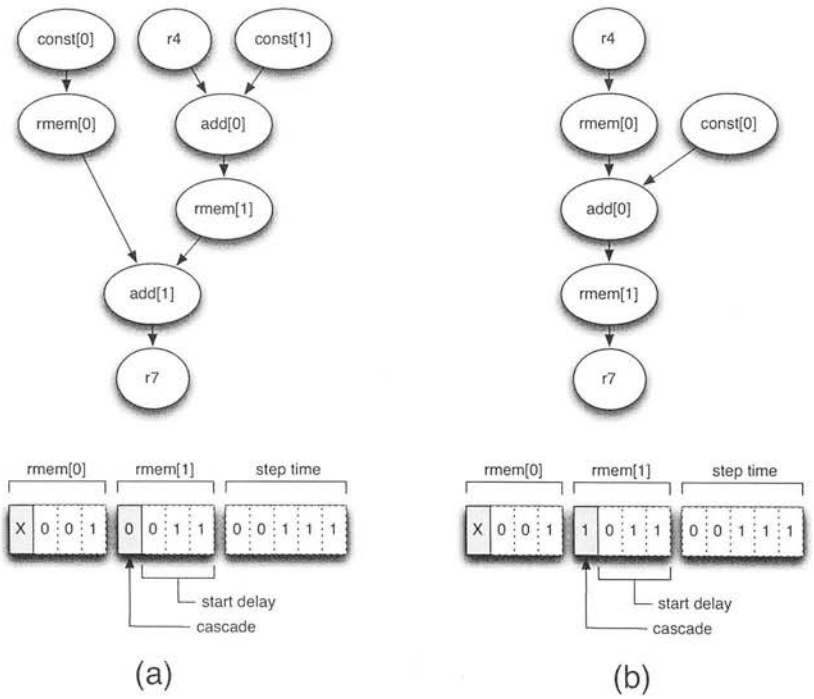


Figure 4.34: Step data flow graphs showing memory read operation cascading. (a) example of one data path containing two independent memory read operations. (b) example of one data path containing a chain of two dependent memory read operations. The section of the configuration word corresponding to the memory access cells and RRC step time, is shown for each, with the cascade bit highlighted.

Figure 4.34(a) shows an example step data flow graph, involving two memory accesses. When these access non-conflicting addresses at run-time, then they can happen in parallel. This is shown in figure 4.35(a), where the memory fetches (shown by the black arrows) overlap in time. If there is a conflict (i.e. they are both accessing the same memory bank), then the arbitration logic serialises them, as shown in figure 4.35(b).

[34] thus dynamically extending the critical path.

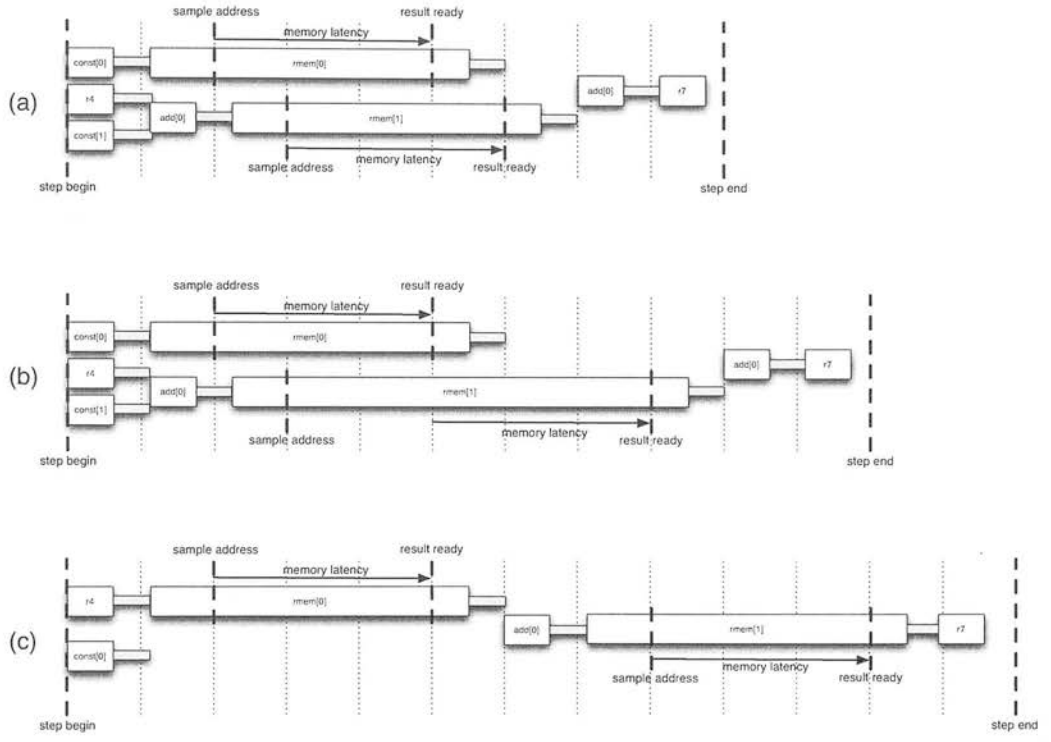


Figure 4.35: Timing diagram (horizontal axis representing time) for the step DFG shown in figure 4.34. RRC clock cycles are shown by dotted lines. Boxes show cell combinatorial delay, grey boxes show interconnect delay, gaps represent idle time. Events are shown in thick dotted lines. (a) Timing for figure 4.34(a) when no memory access conflict exists, (b) timing for figure 4.34(a) when a memory access conflict exists, resulting in additional dynamic delay (due to being queued). (c) Timing for figure 4.34(b) (no conflict possible).

Before pipelining was made possible by the work outlined in chapter 5, dependent memory operations within a single-step kernel were only possible via the addition of hardware support. This is referred to as *cascading*. This allows multiple instances of the `rmem` cell to be involved in the same data path, and be dependent upon each other[35], by ensuring that the address read delay of the cascaded cell is extended by any dynamic delay introduced by queuing during the preceding memory read operation.

Figure 4.34(b) illustrates cascading. Cascading is described in the configuration word by a single bit for each `rmem` cell instance (the cascade bit—shown in grey in the figure), which indicates whether or not the cell is cascaded to the cell with the preceding instance index. `rmem[0]` has no cascade bit. A start delay of 0 means one clock cycle from the reference point, which is either the beginning of the step (if not cascaded to the preceding cell), or when the preceding instance returned its value (if cascaded). The step time field does not include memory latency, as the counter is paused whilst the fetch is happening. This can be seen in figure 4.34, as the bit patterns differ only by the cascade bit, yet the actual timing of the events

[35] i.e. where the read address of one is affected by the value returned by another.

is different in the two cases presented: i.e. the total fetch time and execution time is longer in (b), but the combinatorial critical paths of both are similar. Also, both cases share the same start time pattern for `rmem[1]`, despite the fact that in (b) the fetch will begin much later (due to cascading).

Note that with the introduction of support for internally pipelined cells (section 5.7), data memory can be accessed much more efficiently by pipelining the memory read operations such that the address is given in one step/iteration, and the result is obtained at the beginning of a step/iteration some number of iterations later. This allows the latency of the memory access to be hidden by pipelining the step around the internal pipeline geometry of the memory access operation, thus preventing the iteration interval from being limited by the memory access latency. This can be seen in figure 4.36, which shows the timing diagrams corresponding to those in figure 4.35, but with internally pipelined cells. The total number of cycles per iteration is less in all cases. Note that further improvements in throughput can be achieved by pipelining the step around the internal pipeline geometry of the cells, eliminating the memory idle time.

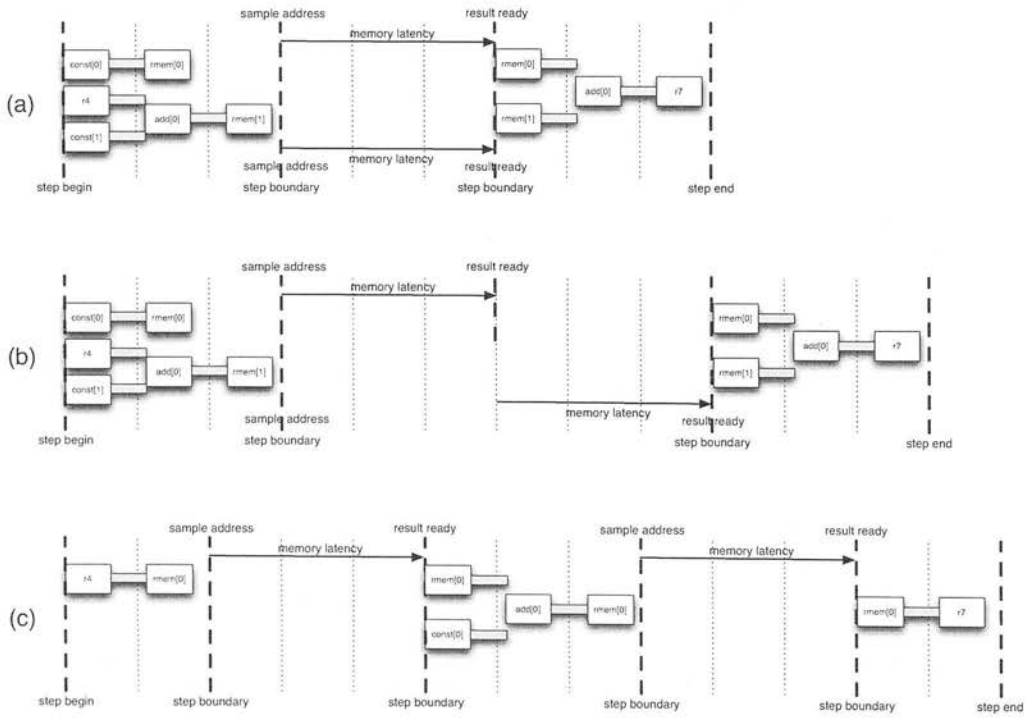


Figure 4.36: Timing diagram (horizontal axis representing time) for the step DFG shown in figure 4.34, with internally pipelined memory access cells. (a) Timing for figure 4.34(a) when no memory access conflict exists, (b) timing for figure 4.34(a) when a memory access conflict exists, resulting in additional dynamic delay (due to being queued). (c) Timing for figure 4.34(b) (no conflict possible).

4.11.2 Contribution: RMEM Cascading Algorithm

An algorithm is needed to determine when RMEM cascading is required, and in what order the `rmem` cells must be allocated. This section proposes one such algorithm. In order for cascading of RMEMs to work, the `RMEM` operations in a dependency chain (all within the same step) must be allocated to `rmem` cells contiguously in asceding order of index, starting with the one at the beginning of the chain.

The approach is to analyse the data paths in each step that involve `RMEM` operations, by removing all other operations from the data paths. These RMEM-only data paths can then be easily inspected to determine which `RMEM` operations depend on which others within the same data path. The order of independent `RMEM` operations within a data path is irrelevant, but dependent operations must be contiguous and in the correct order.

The `RMEM` operations are first re-ordered so that each RMEM-only data path consists of contiguous `rmem` cell instance indexes. This is done by visiting the RMEM-only data paths in turn, in arbitrary order. Then, within each RMEM-only data path, the operations are re-ordered in ascending order of start delay. This means that there will be no RMEM data paths interleaved, and all edges within each RMEM data path will be in propagation order. This satisfies the requirements stated above.

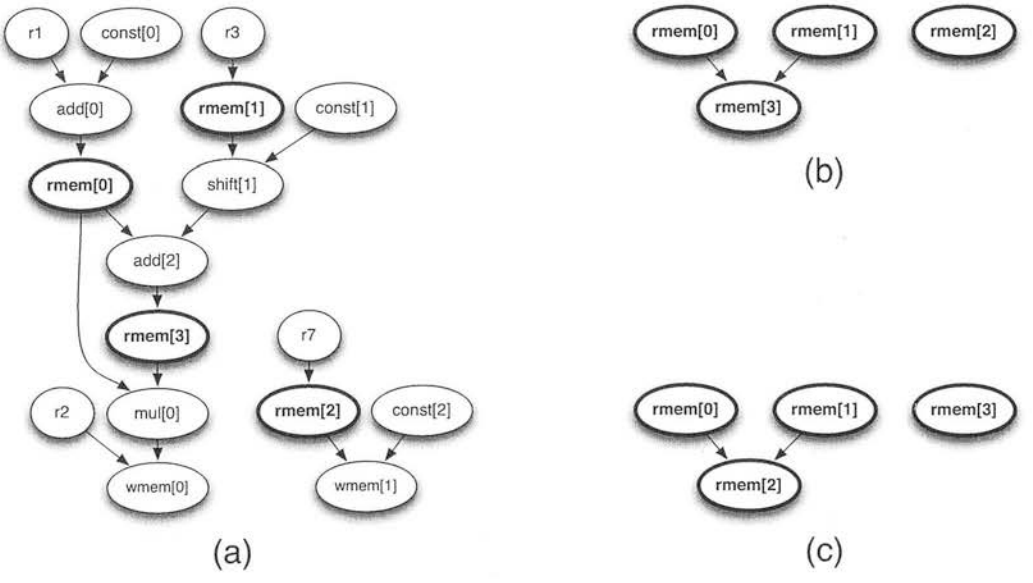


Figure 4.37: Analysis of `RMEM` operations. (a) Original step data paths, with RMEMs shown in bold. (b) Step data paths reduced to only the relationship between `RMEM` operations. (c) RMEM-only data paths with cell instances re-ordered for cascading.

Figure 4.37(a) gives an example, showing the data flow graph for a step containing two data paths, each of which has `RMEM` operations. This is in the form of the data model following parallelisation and resource allocation. As a result, the nodes represent physical cell instances. Figure 4.37(b) shows the same data paths after having removed all other operations, leaving only the RMEMs. The relationship between the RMEMs (i.e. dependencies) can be determined

directly by looking at the immediate predecessors. However, at this point, the relationship is not yet compatible with cascading, since the data paths do not yet contain contiguously assigned `rmem` cell instances. Figure 4.37(c) shows the same RMEM-only data paths with the cell instances re-ordered, which is now compatible with cascading.

It should be pointed out that the instruction cell resources of a given type are allocated in the same order as the DFG edge names, and the edge names are ordered the same as the instructions in the assembly. The nature of the assembly is such that it is impossible for an instruction to appear in the assembly before another instruction that depends on it. This means that RMEMs will already be ordered according to where they appear in the data path that they are part of. However, it is still possible for an independent `RMEM` instruction to have been inserted into the assembly between two dependent `RMEM` instructions, which is why the re-ordering is required. The order shown in figure 4.37(b) is indicative of this interleaving of data paths in the assembly.

With the `RMEM` operations re-ordered (as shown in figure 4.37(c)), it is then possible to determine which cascade bits to enable. If an operation is dependent on another in the same RMEM-only data path, then the cascade bit is enabled. In situations with more than two `RMEM` operations in the same data path, the cascade bit may also need to be set. Essentially, where an `RMEM` operation is dependent on more than one other `RMEM` operation, despite those other `RMEM` operations being independent of one another, the cascade bit should be set on all but the one with the lowest instance index. So in the figure, `rmem[1]` and `rmem[2]` would both have the cascade bit set, despite `rmem[1]` being independent of `rmem[0]`.

These are false positives, but are necessary since the dependent memory read must occur after the completion of all the memory reads on which it depends (including any dynamic delay). There is no way to directly describe this situation in the configuration. The algorithm will have re-ordered these operations into a contiguous set of cell instances, with the cascade bit being set on each of them. This serialises each of the memory accesses (including the independent ones), which achieves the desired effect of taking into account all of the dynamic delays, but also decreases the throughput, since it removes the parallelism. This is a hardware limitation that could be avoided at the expense of additional configuration bits, but since this situation is rare, this was deemed unnecessary.

4.12 Global Register Reallocation Information

The scheduler has no concept of how the interconnect will be configured for each step. This is the task of the routing tool (*mapper*). Therefore, the choices as to which cell instance is assigned to which task in each step, may not be the most efficient in terms of physically mapping to the array—i.e. may have longer paths than necessary. Only the routing tool has the information needed to optimise the allocation of which active operation of each type in each step maps to which instance of the physical cell of that type. The routing tool therefore requires the freedom to change this allocation. This section first proposes several methods for how the mapper tool could improve routability by reallocating registers. Then an algorithm is proposed for generating the information as to which registers are connected across step boundaries. Finally, a method for how the mapper would use this information is proposed.

Reallocating stateless instruction cells is no problem, and can be done arbitrarily without affecting the behaviour of the program. However, cells that have state must be reallocated globally, so that the information stored there is consistent between uses. Registers are the most common example of cells that maintain state. Due to their prevalence—several in almost every step of the program—the allocation of registers has a significant effect on the performance of the program,[36] and on the routability.[37]

Figures 4.38 and 4.39 demonstrate this by example. The white boxes represent the steps of the program, with the thick grey arrows showing control flow between them. The program consists of two loops, one with a single step (i.e. a kernel), and one with several steps, plus the usual program entry and exit steps, with some computation being performed prior to entry to the first loop. This is best seen in figure 4.38(a). The majority of computation is done in the two loops.

There are 4 registers in this example, which pass information between the steps. The registers are shown in figure 4.38(b) on entry to and on exit from each step, with dark vertical lines (and loops) showing the lifetime of the information stored there. Each information line has a number written beside it,[38] which indicates the globally unique index of that piece of information. When a line originates from the output of a step (the bottom side of the box), that indicates that the information is created inside that step. When a line ends at the input of a step (the top side of the box), that indicates that the information is last read from in that step, and the register will either be dead on exit (no line on exit), or have a new piece of information written to it (another line begins on exit). When a line passes through a step, that indicates that the same information is still stored in that register—i.e. it is not overwritten in the step. A small coloured box is shown beside each step, indicating the amount of congestion present in that step. This congestion is assumed to be due to the combination of how busy a step is and excess lengths in the paths to and from the registers.[39]

[36] by their effect on the critical path.

[37] due to congestion resulting from several long paths becoming tangled up.

[38] at the moment when the information is created.

[39] as will be the case in typical programs.

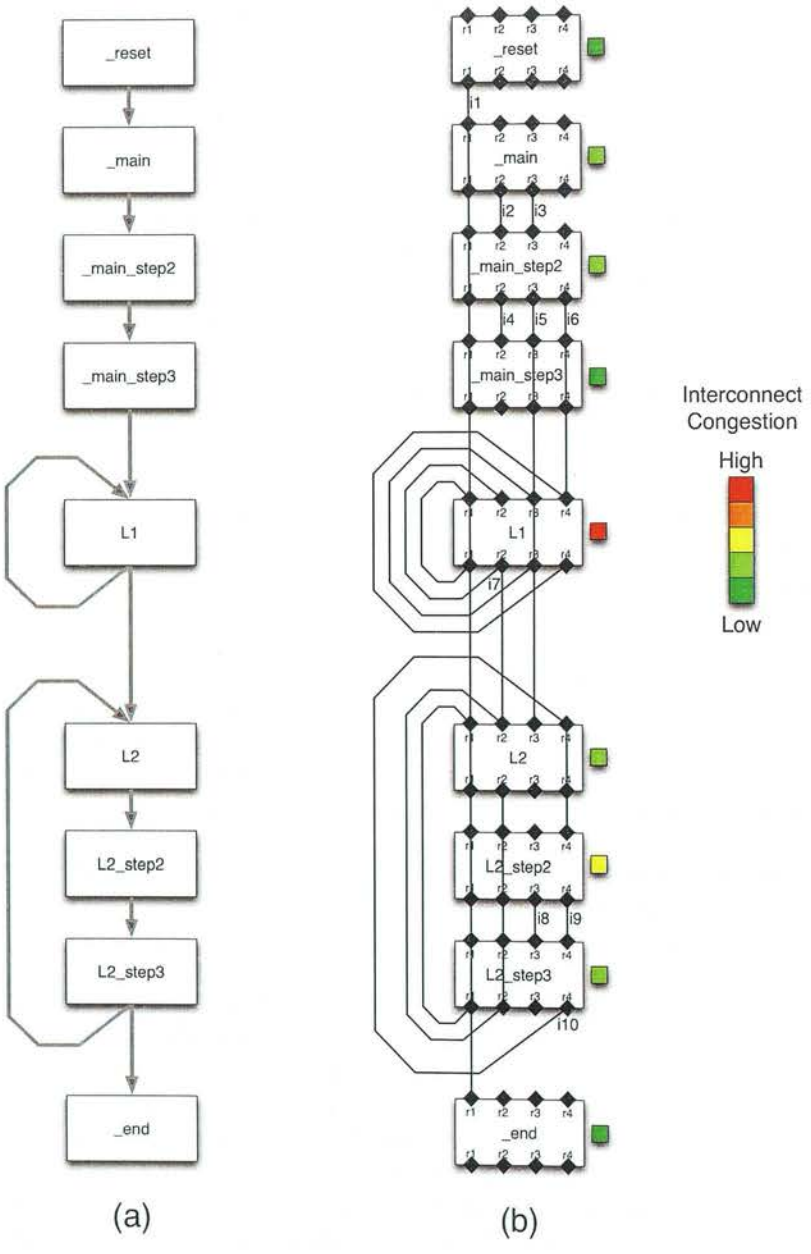


Figure 4.38: Information flow diagram for a simple program. (a) Program steps and control flow. (b) Information flow between registers, with original register assignment made by the scheduler.

Figure 4.39(a) shows the program with the original register allocation assigned by the scheduler[40], and is identical to figure 4.38(b) but with some added highlights. The information assigned to register $r3$ by the scheduler is shown in blue, and the information assigned to reg-

[40] which has no notion of routability.

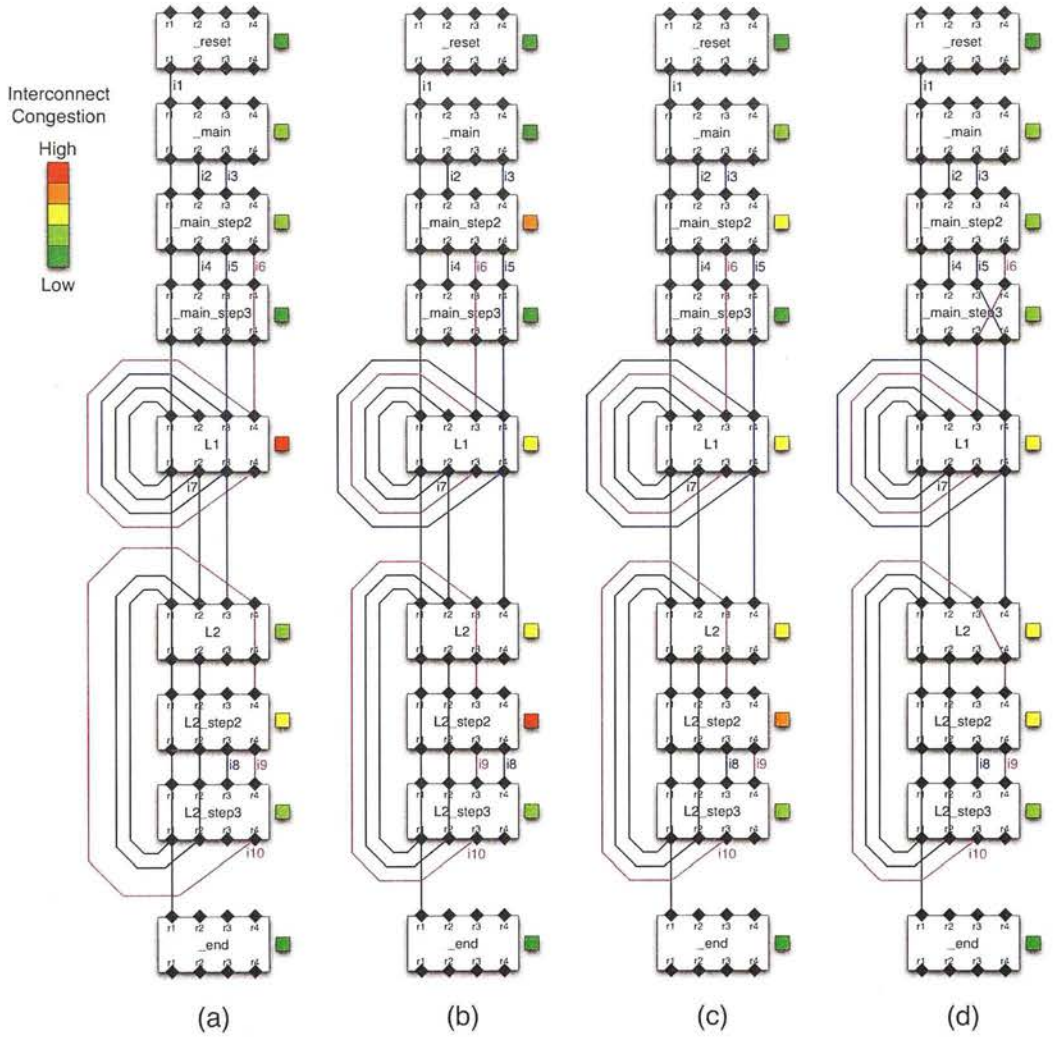


Figure 4.39: Information flow diagram for the simple program in figure 4.38. (a) Original register assignment made by the scheduler. (b) Register assignment following the global swapping of registers $r3$ and $r4$. (c) Register assignment following a reassignment of the register associated to each piece of information originally mapped to $r3$ and $r4$. N.B. Each piece of information still has one particular register holding it throughout its entire life time. (d) Register assignment with the addition of register-to-register copies (limited to $r3$ and $r4$), allowing pieces of information to be moved to different registers.

ister $r4$ by the scheduler is shown in purple. This information is moved around as a result of reallocation. For clarity, figure 4.39 and associated commentary will only consider the effects of reassignment between registers $r3$ and $r4$. However, it should be noted that reassignment of all four registers would be possible, and could lead to significant improvements in routability.

4.12.0.1 Global Register Swap

In the original register assignment made by the scheduler (figure 4.39(a)), we can see that the kernel `L1` is heavily congested. Other steps in the program are less congested, so one approach would be to globally swap one register for another such that the congestion is minimised in `L1`. To this end, figure 4.39(b) shows the result of swapping registers $r3$ and $r4$ globally, to reduce the congestion in `L1`. The information originally assigned to $r3$ is highlighted in blue, and the information originally assigned to $r4$ is highlighted in purple.

Globally swapping one register with another[41] can improve the situation for particular steps, but due to the substitution being global, the same allocation may be problematic in other steps. In figure 4.39(b), the global reallocation has had a positive effect on the routability of `L1`, and incidentally also on `_main`, but has had a negative effect on the routability of `_main_step2`, `L2` and `L2_step2`. `L2_step2` is a particularly busy step, so the overall effect of the global reassignment in this example was to make the situation worse.

4.12.0.2 Global Information Swap

Even if an algorithm were devised to somehow provide a compromise across all the steps in the program, the allocation resulting from a global re-assignment could still be sub-optimal in many steps—potentially leading to congestion or excessive critical path—and the problem becomes increasingly worse with program size[42].

A purely global reallocation is unnecessary. The role of registers is to pass information between steps. So long as the same register is used on both sides of the step boundary over which the information is passed, the program behaviour remains the same. Therefore, it should be possible to swap two registers in only the steps where those registers represent the same piece of information. This *information swapping* approach is demonstrated in figure 4.39(c). It can be visualised as follows: as introduced earlier, each piece of information is represented by a continuous vertical bar (with possible circular feedback) passing through the same register on each step that it passes in to and out of. Each of these bars can be dragged horizontally from one register to another, without affecting the program behaviour.

In this example, only the information previously assigned to $r3$ (i6) and $r4$ (i5) in the most heavily congested step, `L1`, have been swapped. This affects the steps where this information also exists, i.e. `_main_step2` (on exit only), `_main_step3`, `L2`, or where this forces another piece of information into another register, i.e. `L2_step2` (on entry only). The effect is an improvement in the routability of `L1` (the primary objective), and a slight worsening in the routability of `L2` and `L2_step2`. In this example, `L2_step2` has not been affected so badly this time, due to the extra congestion imposed by a global swap being the result of the data paths writing to $r3$ and $r4$ on exit from `L2_step2` (i.e. information i8 and i9), which become longer if those registers are swapped. Similarly, `_main_step2` has not been affected so badly this time, due to the extra congestion imposed by a global swap mostly being the result of the data path reading from $r3$ (i.e. information i3), which becomes longer if $r3$ and $r4$ are swapped. The net effect is an overall improvement in routability; with no heavily congested steps.

[41] as is done for other cells that maintain state.

[42] in terms of number of steps.

4.12.0.3 Information Cross-over

A further improvement is possible: the information could be moved from one register to another during its lifetime, allowing the information to be transferred to a register that is in a more optimal location in a busy step, with the move happening in a less busy step[43].

Figure 4.39(d) shows the effect of this final optimisation. Here the blue and purple information bars can be seen to have been bent in certain steps, where the information has been transferred to a different register inside the step. This occurs in `_main_step3` and `L2`. The transfer of i5 and i6 in `_main_step3` allows `_main_step2` to be restored to the original register assignment, which results in less congestion in that step—but at the expense of a small reduction in routability of `_main_step3`. Similarly, the transfer of i10 in `L2` reduces the congestion in `L2_step2`, at negligible cost in `L2`. The net effect is a further improvement in overall routability.

The next sections describe a method to obtain the information needed to perform both of these partial register reallocation schemes: global information swap or information cross-over. This is referred to as *Global Register Reallocation Information*.

4.12.1 Contribution: Obtaining The Global Register Reallocation Information

Prerequisites: Live register information at the basic block level, and temporary register assignment information for each step that was constructed from the basic blocks.

Results: The list of individual pieces of information entering and exiting each step in the program via each register.

The control flow graph and live register information can be used to determine which input and live output registers represent the same piece of information. This information can then be used by the routing tool to allow registers to be reallocated so that they appear closer on the array to the cells to which they are connected. This significantly reduces pressure on the interconnect,[44] avoiding congestion, and decreasing the propagation delays.

The information obtained during scheduling/parallelisation (section 4.8) defines the lifetimes of each piece of information (more specifically, each DFG edge—see section 4.5) between the steps derived from each basic block. This then augments the live register information obtained using the live register identification algorithm (presented in section 4.7), which defines which registers actively store information between basic blocks. By combining the two, it is possible to determine the lifetime of each piece of information across all possible execution paths in the entire program.

In other words, after scheduling, it is possible to equate each active register on input and output of each step, to a globally unique piece of information. Each piece of information is assigned a globally unique ID.

[43] where the overhead of the connection between the two registers will not cause problems.

[44] since fewer interconnect edges are needed.

For each junction between basic blocks in the program control flow graph (CFG), the live register information is used to determine which output registers and which input registers represent the same piece of information being passed between those basic blocks, as shown in figure 4.40. A similar thing is done for the temporary registers between each step in a basic block. The latter is much easier, since the steps in a basic block are always executed in sequence (i.e. are always an example of figure 4.40(a)).

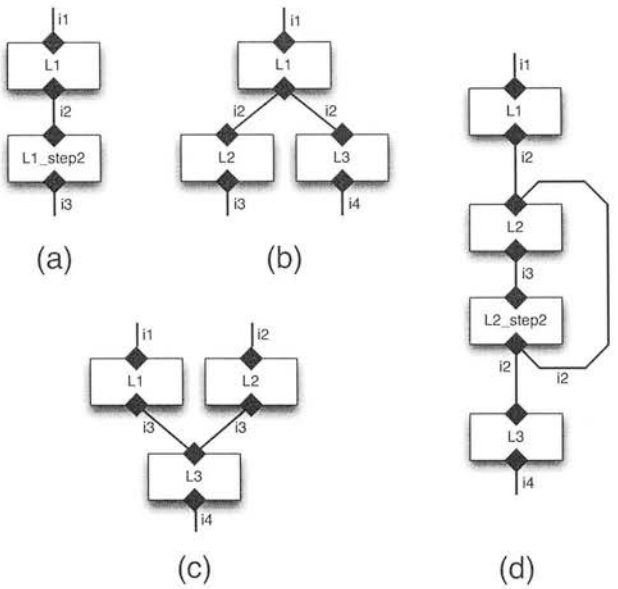


Figure 4.40: How control flow defines on which step boundaries a given register represent the same piece of information. (a) Linear control flow (e.g. between steps derived from the same basic block). (b) Control flow branch (e.g. conditional jump). (c) Control flow join (e.g. return from function). (d) Loop.

It should be noted that it is safe to underestimate the number of unique pieces of information present in the program, but it is dangerous to overestimate it. Underestimating results in registers being bound together in more steps than is really needed, whereas overestimating results in values being corrupted[45].

For this reason, the presence of loops has to be carefully taken into account. In purely sequential control flow, when a given register is overwritten, this results in it storing a different piece of information on entry and exit. However, if that same step sequence appears within a loop, then the control flow path back to the beginning can mean that these are in fact the same piece of information. This can be seen in figure 4.40(d), where information on entry to `L2` and on exit from `L2_step2` have to be the same piece of information. However, that information is being updated within the loop. A common example of this is the loop counter. These are defined to be the same piece of information, since the same register must be used at the beginning and end of the loop in order for the new value to be seen correctly in the next iteration.

[45] since the register could be changed across a boundary where the same register should really be used.

4.12.2 Contribution: Using The Global Register Reallocation Information

The routing tool can use the global register reallocation information to re-assign register numbers, in order to minimise interconnect path lengths. Reallocation consists of looking at each boundary between steps where one step passes control to another. At that boundary, a single register must be assigned to each piece of information that needs to pass across that boundary. The register may be chosen from the set of registers available across that boundary. Since each step may pass control to more than one other step,[46] the assignment must be consistent for all boundaries involving the same step.

4.12.2.1 Method 1: Global reassignment

The order in which the boundaries are examined is significant. The search space is potentially quite large, and the process of calculating the cost is expensive[47]. Therefore, a sensible heuristic is needed.

The proposed heuristic approach consists of first applying the global information swap technique, followed by information cross-over. The steps are visited in descending order of complexity, such that the steps that are likely to be the most congested are dealt with first, thus having the most freedom to reallocate. Register reallocation is performed on the boundaries involving each step, in this order. When a piece of information has been reallocated to a different register, that allocation (information to register) is frozen in all steps in which that information exists. Each step is only allowed to reallocate registers for pieces of information that haven't yet been frozen. This corresponds to the global information swap technique.

If a piece of information is frozen, but reallocating it to a different register in the current step would be advantageous, it may be possible to move it to a different register, if the control flow allows. This corresponds to the information cross-over technique. Information cross-over is normally applied only on step boundaries within the same basic block, as these are usually the only boundaries that have linear control flow. Other forms of control flow (loops) require that the move is performed in other steps too, which makes the problem more complex to solve.

4.12.2.2 Method 2: Register renaming

A simpler technique to implement takes advantage of the fact that most of the execution time of a well-mapped program is spent in kernels. Therefore, optimising these alone should be sufficient. Also, kernels have the highest core utilisation, so will benefit most from a less constrained allocation. The neighbouring steps to kernels tend to be very simple, performing tasks such as initialising the loop iterator and addresses. So, adding complexity to these neighbouring steps will have little impact. The control flow is also very simple: the step before the kernel always passes control directly to the kernel, and the kernel always passes control either to itself or, once finished, directly to the step after the kernel.

[46] depending on the state of the machine at the time of execution.

[47] due to allocation and routing needing to be performed on each attempt.

The technique involves de-coupling information in the kernel from the steps before and after the kernel. Any information that enters the step before the kernel must be transferred to a different register when exiting that step (where it enters the kernel). Similarly, any information that exits the kernel must be transferred to a different register when exiting the step after the kernel.

After this de-coupling, the registers in the kernels can then be reallocated freely. These registers are then locked in place, but this will only affect the steps before and after the corresponding kernel. The cost is therefore increased register-to-register activity in the steps surrounding each kernel—roughly half of which cannot be reallocated, but the kernels get a completely free allocation, which leads to a significant net win.

4.13 Results

This section shows the results from experiments that were devised to demonstrate the following:

Section 4.13.1: The ability of the tree follower scheduling algorithm (section 4.9.2) to generate configuration contexts out of basic blocks, and their quality—in terms of resource utilisation/parallelism, and execution time.

Section 4.13.2: The number of additional registers that live register identification (section 4.7.1) makes available for scheduling and other purposes, and the effect this has on the quality of the resulting schedule.

Section 4.13.3: The effectiveness of the register starvation avoidance schemes (section 4.10) in allowing complex basic blocks to be scheduled for highly resource starved cores, and the quality of those schedules.

Section 4.13.4: The effect of register renaming (section 4.12.2) on connection lengths, when applied to configuration contexts with high utilisation.

4.13.1 Results: Scheduling Algorithm

This section looks at the performance of sequences of configuration contexts generated from a given basic block by the scheduling algorithm presented in section 4.9.2 on page 92, when subjected to certain resource constraints.

The experiment is designed to take a kernel occupying a significant fraction of a small RICA core, and artificially restrict the availability of certain key resources, to force the scheduling algorithm to split the kernel up into multiple steps. The quality of the resulting schedule is then analysed, in terms of effect on step count, total critical path, and throughput.

The example used is a 2-D discreet cosine transform filter (8x8 DCT-II) [80] common in JPEG/MPEG image compression. The implementation of this filter (in C) performs the 1-D DCT as a single loop (kernel), which is called twice—once to operate on the columns, and once on the rows. The compiler generates a single basic block for this inner loop, the resource requirements for which are shown in table 4.6. The scheduler processes all of the basic blocks generated by the compiler, however this analysis will only look at this kernel, as it should represent the majority of the execution time.

As the base line, the scheduler is given a target processor with sufficient resources for the entire kernel to fit into a single step. The experiment then involves selecting some key resource (instruction cells) types, and gradually reducing their availability (instance count). Properties of the resulting configuration contexts are then analysed.

Resource	Instance count
add/comp	43
constant	29
logic	0
multiply	16
multiplexor	0
shift	0
jump	1
read memory	8
write memory	8
register	35

Table 4.6: DCT kernel resource requirements, in terms of instruction cells on the target architecture.

The following resources were constrained in turn: multipliers (MUL), memory read (RMEM), memory write (WMEM), addition/comparison (ADDCOMP). For each series, only the corresponding resource is constrained—all others are available in sufficient quantity. The instance count is then ramped down, to show the resulting gradual degradation in terms of execution speed (throughput—figure 4.43) and program memory cost (step count—figure 4.41). The theoretical minimum step count can be calculated as follows:

$$steps_{min} = ceil\left(\frac{n_{required}}{n_{available}}\right)$$

where n represents instance count of the constrained cell type. In all cases tested, the scheduling algorithm meets this theoretical minimum.

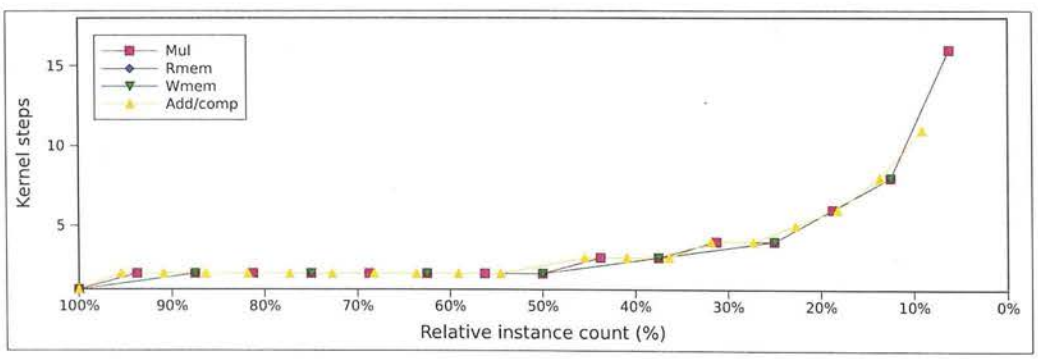


Figure 4.41: Step count resulting from multiple runs of the scheduling algorithm on the DCT kernel, against availability of certain key resources. Starving the kernel of computation resources makes it map to multiple steps, to time-domain multiplex the available resources. In each case, the scheduling algorithm produces the minimum number of steps possible with a given constraint (e.g. 10% availability ⇒ 10× fewer cells than needed ⇒ a minimum of 10 steps required).

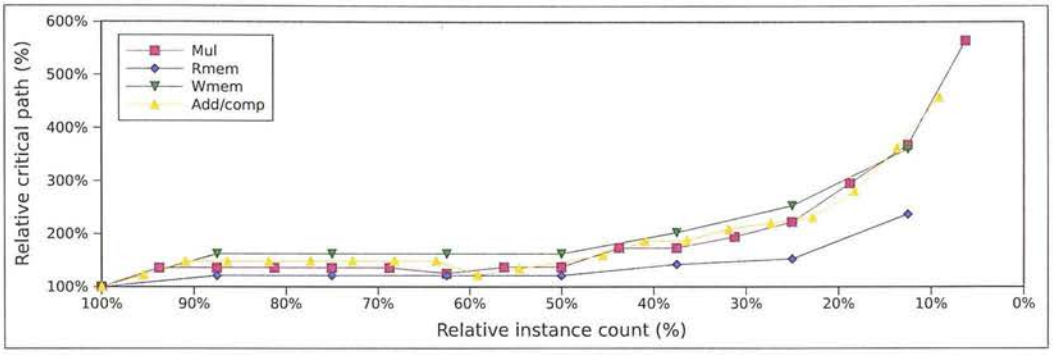


Figure 4.42: Total critical path resulting from multiple runs of the scheduling algorithm on the DCT kernel, against availability of certain key resources. The total critical path is the sum of the critical paths of each of the steps produced. The increase indicates how much parallelism has been lost as a result of partially serialising the data paths into steps. The scheduling algorithm manages to keep this well below what would be expected from the increase in the number of steps, e.g. a 10× reduction in `addcomp` cell availability results in a 10× increase in steps (see figure 4.41), but only a 4.5× increase in critical path.

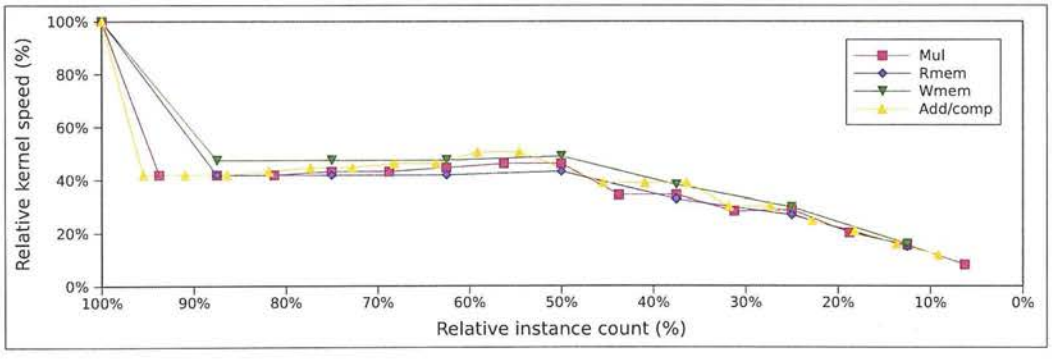


Figure 4.43: Throughput resulting from multiple runs of the scheduling algorithm on the DCT kernel, against availability of certain key resources. Throughput is based on the total measured execution time of the kernel, and includes the effect of step loading times. The decrease in throughput closer matches the increase in the number of steps, rather than the increase in critical path. For example, a 10× reduction in `addcomp` results in a 10× increase in kernel steps (figure 4.41), but a 4.5× increase in total critical path (figure 4.42), leading to a 10× reduction in throughput. This is because the step load-time dominates in this example.

Additionally, figure 4.42 shows the sum of the critical paths of the resulting steps, which gives a clearer view of what the scheduling algorithm has done. This is related to execution speed, but execution speed is also further affected by the step loading times imposed by the additional steps. The throughput (figure 4.43) and total critical path (figure 4.42) graphs are normalised to the base line (with no resources constrained).

The degree of overlap achieved is shown in figure 4.44. This shows the relative increase in critical path v.s. the relative reduction in resource availability. An overlap of 0% indicates a

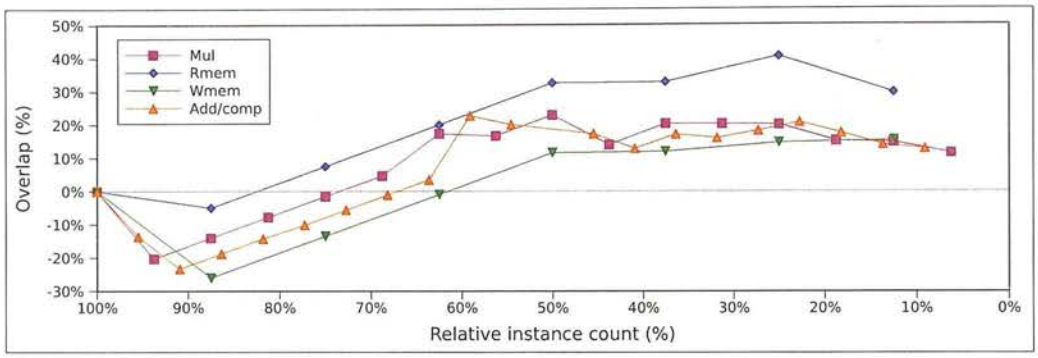


Figure 4.44: Achieved extent of overlap of the critical path, against availability of certain key resources. Overlap is the increase in total critical path v.s. the decrease in resource availability, both relative to the base line (unconstrained case). An overlap greater than zero indicates a net saving. The relative overlap is less than zero when the resources are only slightly limited, indicating that some of the data paths that ideally would run in parallel are being separated out into a second step. This is unavoidable. The relative overlap improves as more of the data paths have to be brought into the second step, creating a more balanced schedule. When more than two steps are necessary, there is a lot more room for the scheduling algorithm to improve parallelism, shown here by a positive relative overlap.

linear relationship. A value greater than zero indicates that the resource constraint is being absorbed, partially hiding its cost. It is calculated as follows:

$$overlap = \frac{cp_{baseline}}{cp_{total}} - \frac{n_{available}}{n_{required}}$$

where cp represents critical path, and n represents instance count of the constrained cell type.

Looking at the data flow graph for the basic block (figure 4.45), all the operations are part of the same data path. Considering the case of the multiplier (MUL), there are 12 instances of this operation with a start delay of 3.46ns (3rd row in the DFG), 2 with a start delay of 5.45ns, and 2 with a start delay of 11.94ns. When less than 12 instances of the multiplier resource are available, the 12 operations in the 3rd row cannot all be performed together. These lie on the critical path, and so splitting the basic block into steps will have to increase the total critical path. If 8 multipliers are available (i.e. half the total required number), the lowest achievable total critical path will occur if 8 of these 12 MUL operations are put in the first step, and the remaining ones put in a second step along with the later MUL operations. The MUL operations should lie at the end of the critical path of the first step (feeding their output into a temporary register), in order to minimise the critical path of the first step. The second step will have the same critical path as the original DFG.

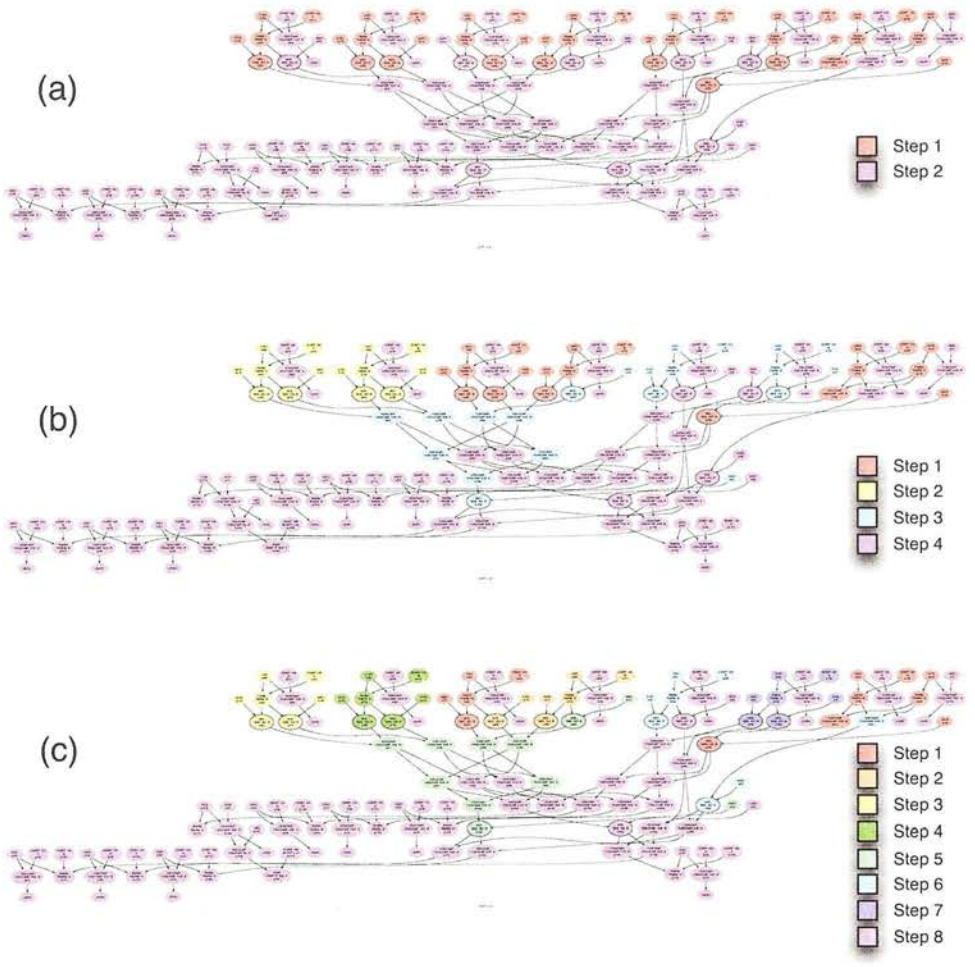


Figure 4.45: Data flow graph of the main kernel (L4) in the DCT example, highlighted according to which step each operation gets placed in. Three situations are shown, with the multiplier (MUL) resource constrained to: (a) 8 instances (2 steps), (b) 4 instances (4 steps), (c) 2 instances (8 steps). Instances of the constrained resource type (MUL) are shown with a black outline.

Putting this into numbers: the 12 MUL operations earliest in the basic block DFG appear at 3.46ns, and produce their output at 4.35ns. Adding a register to store the result, gives an additional 1.44ns for interconnect delay and 0.1ns internal delay for the register. Therefore, the minimum critical path of the first step is $4.35 + 1.44 + 0.1 = 5.89$ns. The entire basic block DFG has a nominal critical path of 16.82ns—as will the second step. Therefore, the minimum possible increase in critical path is $5.89/16.82 = 35\%$.

The actual increase achieved by the scheduler was 37% in this case, which is slightly worse than the theoretical minimum calculated above. Looking at the resulting schedule (figure 4.45(a)), we can see why: the algorithm has chosen to include one of the 2nd level multipliers in the first

step, artificially increasing the critical path. This is because the algorithm used here searches the predecessors of the active edge in arbitrary order. It should perform better if the predecessors were visited in order of their output delay. The tool doesn't implement this approach because this imposes a significant increase in the execution time of the scheduling algorithm. In its commercial use, the tool execution overhead was deemed more significant than the effect on the quality of schedule achieved.

4.13.2 Results: Live Register Identification

This section demonstrates the effectiveness of the live register identification algorithm (described in section 4.7.1 on page 77) in increasing the number of registers available for use by the scheduler. The scheduler infers additional registers (termed *temporary registers*) when a basic block has to be split into multiple configuration contexts (section 4.8.1 on page 84). It also infers additional registers for use in certain assembly-level optimisations, such as converting stack local variables into registers, or counter replication.

This is demonstrated using two examples from elsewhere in the thesis: a 2-D 8x8 DCT transform, introduced in section 4.13.1, and a gamma correction module test bench, introduced in section 5.8.2 on page 174. The DCT example was chosen as it has a rather simple kernel, showing the lower-end of what live register identification will work on. The gamma correction module was chosen as a more complex example, large enough for the compiler to have re-used a lot of registers, making fewer registers available for use as temporaries.

Figures 4.46 and 4.47 show the results for the DCT example, and figures 4.48 and 4.49 show the results for the gamma correction module example. The graphs show the basic blocks in the corresponding program along the x-axis, referred to by index in the program. The bars show the number of instructions in each basic block, to indicate their complexity. The lines show the percentage of all registers in the core that are available for use by the scheduler. The remaining registers were either used by the compiler, or are not known to be safe to use.

The DCT example was compiled and scheduled for a core with 64 registers, 16 of which are reserved for scratch. This gives just enough registers for the compiler to produce uncompromised data paths and blocks. Similarly, the gamma correction example was compiled and scheduled for a core with 250 registers and 35 scratch registers, for the same effect. This minimises the advantage of live register identification, leading to a more fair comparison. The more registers there are in the core (not reserved as scratch), the higher the potential advantage. This is because the number of registers used by the compiler will remain constant, so the ratio of unused to used registers will increase, and the number of which are unused is not known without live register identification.

Without live register identification, when basic blocks are split into steps, temporary register assignment (section 4.8.1 on page 84) can only assign temporary registers from the pool of registers that are active (i.e. written to or read from) in the basic block, plus scratch registers (which are reserved for use by the scheduler). All other registers might contain data that must be transported through the basic block for use later in the program. The effect of live register identification on this is to determine which of these other (inactive) registers might contain important information, making all the others available for use in storing temporaries.

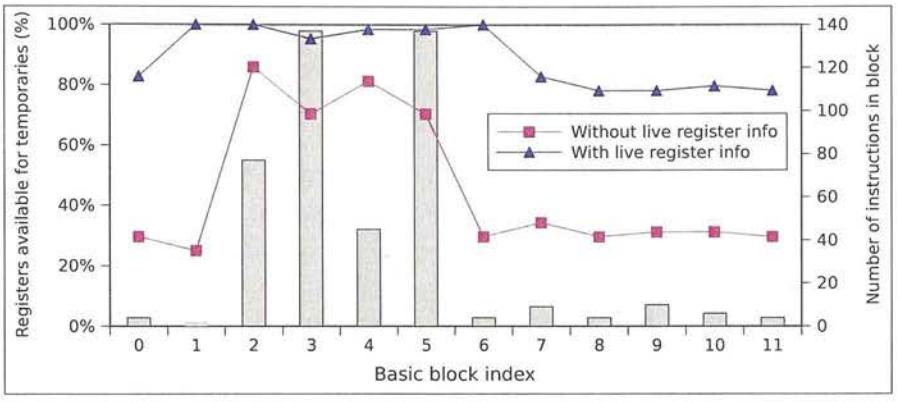


Figure 4.46: Registers available for storing temporary values inside each basic block in the DCT program, with and without live register identification. The bar graph shows the complexity of each basic block (i.e. number of instructions). Without live register information, only the scratch registers and those registers used by the compiler in that basic block are available for use as temporaries. The number of registers used by the compiler is roughly proportional to the complexity of the basic blocks in this example, as shown by the 'without live register info' graph. Any registers not used by the compiler might store important data across that block (i.e. are dormant), and thus can't safely be used. Live register identification determines which of these really are dormant, and makes the rest available for use as temporaries. This mostly benefits the least complex blocks, as the compiler uses fewer registers there.

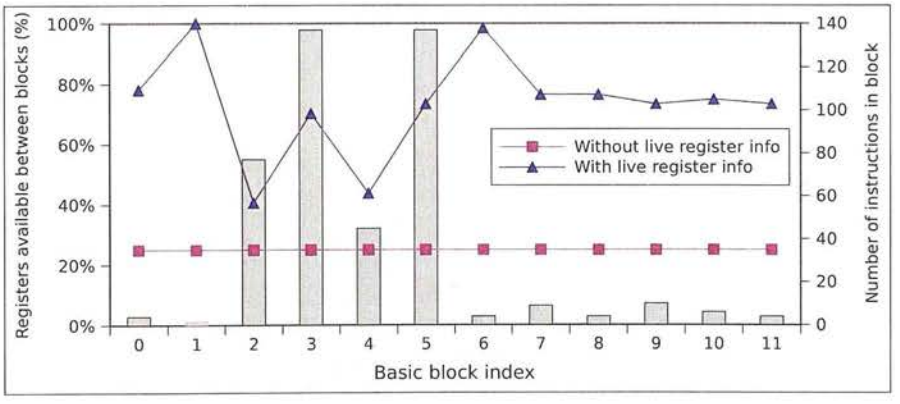


Figure 4.47: Registers available over the boundaries between basic blocks in the DCT program, with and without live register identification. The bar graph shows the complexity of each basic block (i.e. number of instructions). Only the scratch registers are safe to use between basic blocks, as any register written to by the compiler could pass data between basic blocks. Live register identification determines which registers written to by the compiler actually store values across that block boundary, which is generally only a small subset of those written to, as can be seen here. Those which are not live on exit can be mapped entirely to wires by the scheduler (i.e. do not require temporaries), which particularly benefits the most complex blocks.

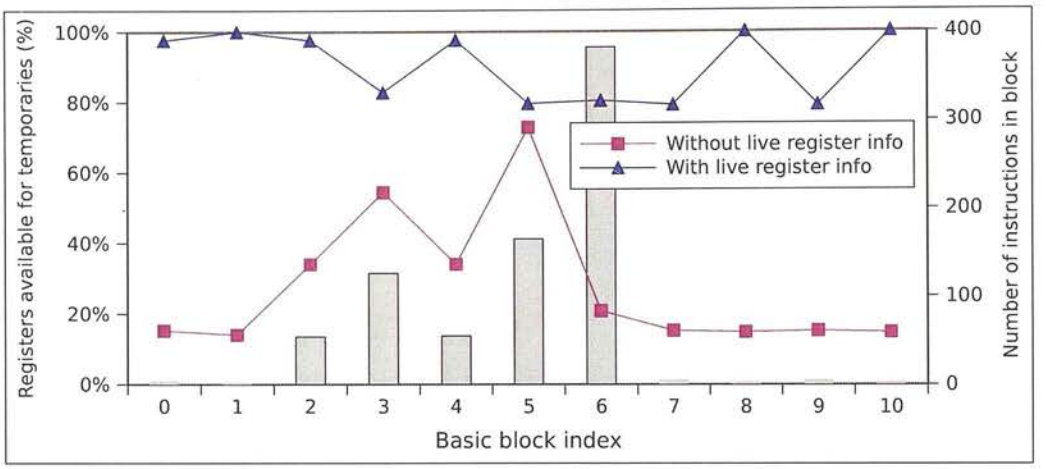


Figure 4.48: Registers available for storing temporary values inside each basic block in the gamma correction program, with and without live register identification. The bar graph shows the complexity of each basic block (i.e. number of instructions). The situation is similar to that in figure 4.46, except that the main kernel (block no. 6) in this example has a high degree of register re-use. Live register identification in this case makes a significant improvement in register availability for temporaries.

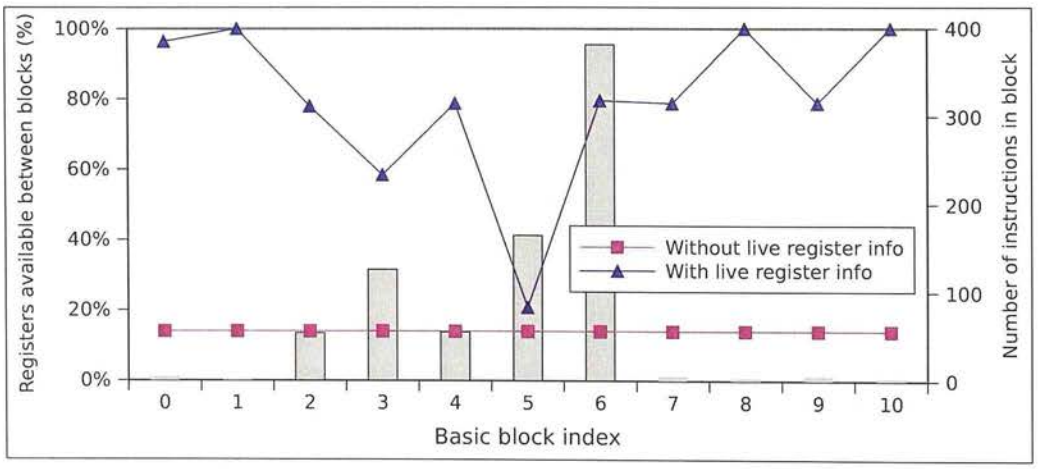


Figure 4.49: Registers available over the boundaries between basic blocks in the gamma correction program, with and without live register identification. The bar graph shows the complexity of each basic block (i.e. number of instructions). The situation is similar to that in figure 4.47—most registers unused by the compiler do not store information between the basic blocks of the program.

Figure 4.46 shows a fair improvement in register availability for use as temporaries in most blocks. The improvement is least significant in the basic blocks with the highest utilisation, e.g. the two kernels: L4 (index 3) and L6 (index 5). It is in these blocks where temporary registers are going to be of the most use. However, even a small increase in register availability can make it easier for the scheduler to form a more efficient sequence of steps. Also note that

the advantage increases linearly as the total register count is increased. The gamma correction example (figure 4.48) paints a different picture: the kernel in this example (index 6) shows a significant increase in register availability. The results in section 4.13.3 show the effect that register availability for temporary registers has on the ability to schedule, and on the quality of the results.

For register availability between basic blocks, without live register identification, the DFG analysis (section 4.5 on page 67) must assume that all registers that the compiler could use store important data. As a result, only scratch registers (reserved for use by the scheduler) can safely be used between basic blocks. Live register identification improves this situation in several ways: it determines which of the register that were inactive in the basic block store important data through that block (i.e. *dormant registers*), it determines which of the block's input registers need to have their value preserved for use later in the program, and it determines which of the registers that were written to in the basic block actually pass information to subsequent blocks. As a result, the improvement is more dramatic in these cases, as can be seen in figure 4.47 and figure 4.49.

Experience shows that the ability to identify which of the inactive registers are dormant makes it possible to eliminate the need for scratch registers. This means that for a given core, the compiler has more registers available to it, which helps it form larger basic blocks which are better candidates for parallelisation.

Another benefit of live register identification—not looked at here—is the effect on reduced activity on the core: By identifying which registers written to in the basic block actually carry data out of the basic block, the number of connections to registers is reduced, which improves routability.

4.13.3 Results: Register Starvation Avoidance

A gamma correction filter [81, 82]—a common module in a typical image signal processing (ISP) pipe [83]—was used to demonstrate the onset and effectiveness of each register starvation avoidance technique described in section 4.10 on page 95. The module performs all of the pixel-level work in a single kernel, which loops over all pixels in the image. This module was chosen since it is large enough to nearly fill the example core, and has several independent data paths (i.e. one for each of the 6 channels processed). This means that there are fewer data dependencies, leading to larger scope for rearrangement. The rearrangement of data paths performed by shuffle is done in a random manner, so this increased freedom should exaggerate any critical path overhead. The register instance count was artificially reduced to an extent that puts significant pressure on the scheduling algorithm, both with and without live register identification, but high enough so that the compiler doesn't map all the local variables to the stack (which would take them out of the scheduler's control).

The compiler's choice of register assignment ensures that register starvation does not occur between basic block boundaries, or internally to the basic block if the instruction stream is followed exactly. Scheduling of the basic block DFG onto the target core turns many of the registers used internally in the basic block into wires. If the core has sufficient resources to perform all the operations of the basic block in one configuration context, then no additional

registers are needed after parallelisation by the scheduler. However, if availability of one or more resources cause the basic block to be split into multiple contexts, additional registers (*temporary registers*) are needed to bring values across the boundaries between steps resulting from that basic block. Therefore, in order to induce the scheduler to use more registers than stipulated in the assembly, the target core must have insufficient resources to map the basic block into a single step.

To this end, the target core was given resources sufficient for the main filter loop except for one resource—`Add/comp`—which was given only 5 instances out of the 61 required. This means that the scheduler must split the basic block into at least $ceil(61/5) = 13$ steps. Since each of these steps must be executed in sequence for every pixel of the image, this severely limits the throughput. This leads to a throughput of a mere 7.2Mpixels/s in the case where there are ample registers, and lower once the registers become constrained.

The experiments involve using the same assembly for each test, with a different register count being given to the scheduler each time. Using the same assembly means that the data flow graph of the kernel remains the same. Therefore, all changes in the resulting netlist will be as a result of the scheduler's actions. When building this assembly, the register count must be chosen carefully. This has to represent the lowest register count to be tried by the scheduler, otherwise the assembly will contain active registers that don't exist in the target architecture.

In order for register starvation to occur at all, the number of internal values needing to be preserved across the internal step boundaries must exceed the number of registers available for this use. However, if the register count is too low, then the compiler will push excessive values on the stack, making the kernel memory bound, causing the memory accesses to dominate the execution time. This would obscure the effect of changes to the critical path. As a fair compromise, a register count of 32 was given to the compiler. Table 4.7 shows the resources in the main loop basic block after compiling with this register count.

Resource	Instance count
add/comp	61
constant	64
logic	24
multiply	12
multiplexor	30
shift	60
jump	1
source	6
sink	6
stream buffer	12
register	11
read memory	1

Table 4.7: Gamma correction filter kernel resource requirements, in terms of instruction cells on the target architecture. Reduced complexity version—with no assembly-level optimisations.

The scheduler was then run several times on this assembly, using a range of different register counts. The range was chosen such that the maximum register count was more than enough for the basic block to schedule without any problem, and the minimum register count is equal to the register count used to generate the assembly. The experiments were repeated with live register identification disabled, so that only the 11 registers active in the main loop (plus an additional 4 scratch registers) are available for use as temporaries. A valid schedule was eventually achieved in each case.

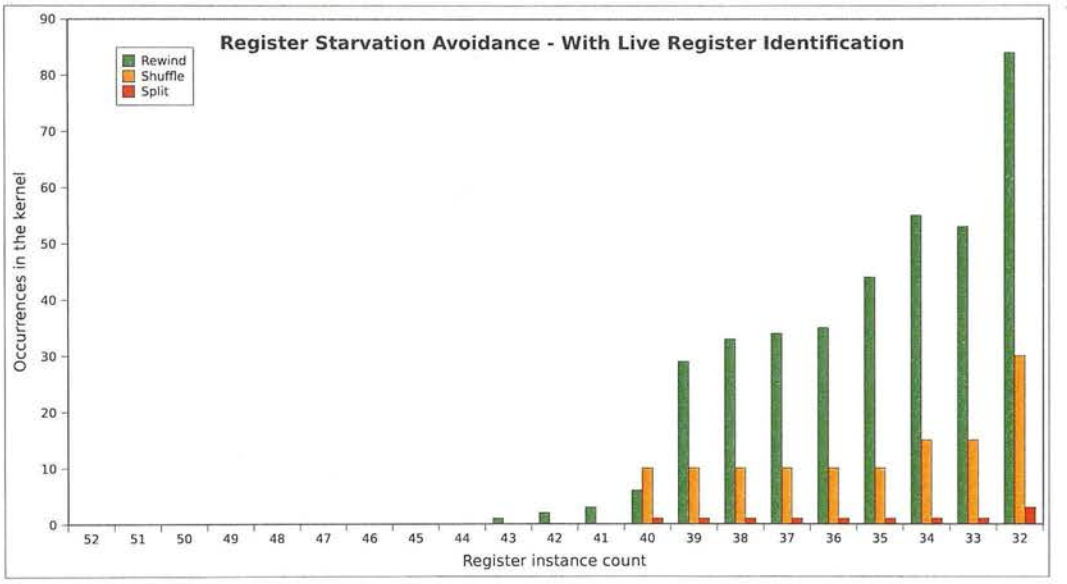


Figure 4.50: Total number of occurrences of each of the register starvation avoidance techniques when scheduling the gamma correction module's main loop basic block, for a range of different register instance counts, with live register identification enabled. In this example, register starvation occurs only when the register count is below 44. As the register resource becomes more constrained, the number of rewind attempts increases, and the more severe avoidance techniques become increasingly necessary.

Figure 4.50 shows how many times each of the register starvation avoidance techniques were instigated when scheduling the main loop basic block. When register starvation is encountered, rewind (section 4.10.1 on page 96) is performed as the first attempt at avoidance. Rewind can be performed several times before a valid schedule is found. If all suitable rewind points have been tried to no avail, shuffle (section 4.10.2 on page 98) is performed next, up to a total 10 times. If a valid schedule still cannot be achieved, basic block splitting (section 4.10.3 on page 100) is performed as a last resort. These techniques are performed as three nested loops, so rewind is tried again after a shuffle has been performed. Similarly, up to a further 10 shuffle attempts may be performed after each split. The bars in the figure show the number of times each technique was tried in total before a valid solution was finally landed upon. As one would expect, the graph shows that scheduling becomes increasingly difficult as the register count becomes more constrained.

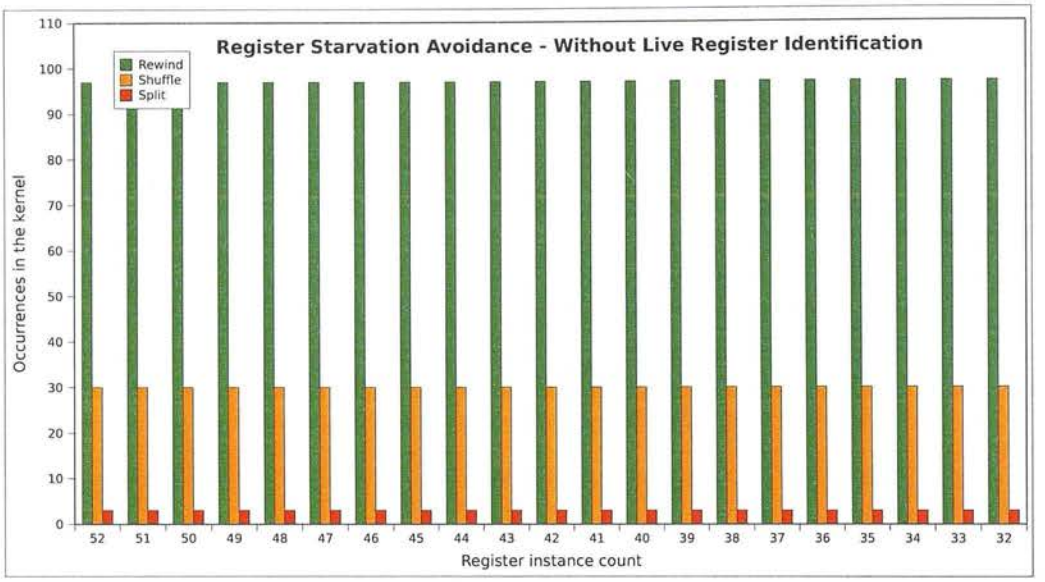


Figure 4.51: Total number of occurrences of each of the register starvation avoidance techniques when scheduling the gamma correction module's main loop basic block, for a range of different register instance counts, with live register identification disabled. Without live register identification, more registers are needed, and thus starvation occurs at a much higher instance count (compared to figure 4.50). All instance counts tried here are deep into starvation (comparable to the most extreme case in figure 4.50), requiring multiple attempts of each avoidance method. Despite this, a valid schedule is obtained from each case.

As a comparison, figure 4.51 shows the avoidance behaviour when live register identification is disabled. The same number of registers are available for use in temporaries irrespective of the total register count, so in each case the scheduling algorithm struggles in a similar manner to the most register starved case when live register identification is enabled.

Figure 4.52 shows how many steps the basic block is scheduled into, in each case. As can be seen, when live register identification is enabled, even in the worst cases only a single additional context was generated, despite the basic block needing to be split several times before a valid schedule could be obtained. Figure 4.53 shows the sum of the step critical paths for the resulting schedule for each register instance count tried. Generally, the data path parallelism seems to stay largely intact, with a maximum increase of 17%. Figure 4.54 shows a similar story, this time measured in terms of throughput. This is based on measured execution time, which includes the step load time overhead. The worst case yields an 11% reduction in throughput. The resulting affect on critical path and throughput doesn't directly correlate with the difficulty in obtaining a valid schedule. This is indicative of the (increasingly) random nature of these avoidance techniques.

When live register identification is disabled, the situation is even more severely constrained, yet register starvation avoidance eventually yields a valid schedule with a 17% increase in total critical path, and a 17% reduction in throughput.

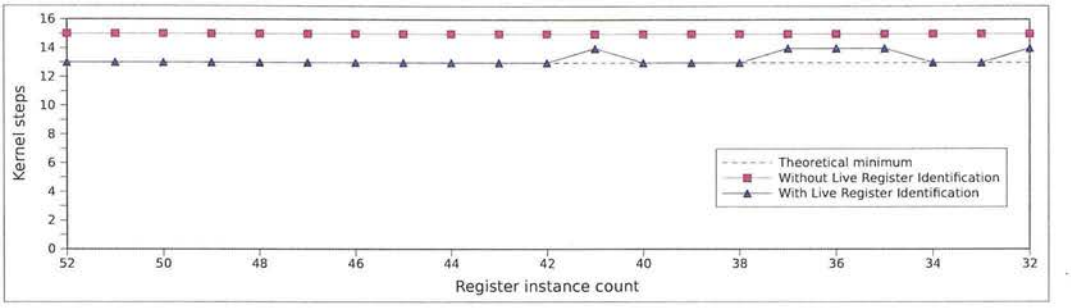


Figure 4.52: Number of steps resulting from the scheduling of the gamma correction module's main loop basic block, over a range of register instance counts. Without live register identification, the difficulty in achieving a valid schedule is similar across all instance counts. This is reflected by the constant overhead in steps. With live register identification enabled however, most cases show no step overhead. Since the rewind attempts are random, and the first attempt to produce a valid schedule is chosen, it is often possible to achieve a better schedule in a more constrained case.

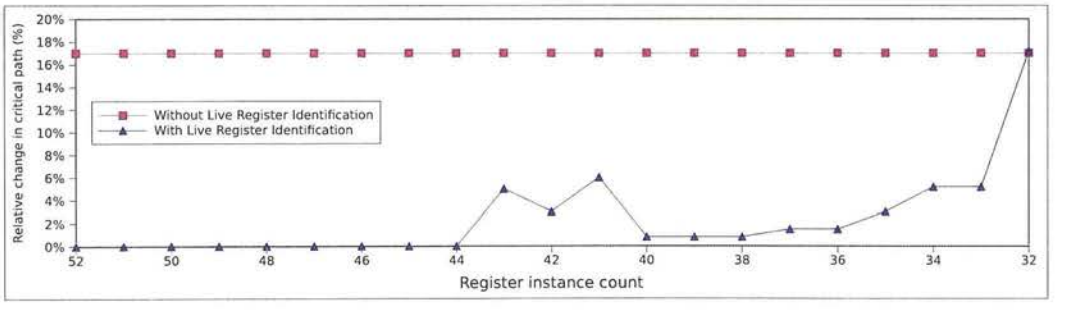


Figure 4.53: Change in total critical path of the resource constrained gamma correction module (compared to the unconstrained case), over a range of register instance counts. Total critical path is the sum of the critical path of each step produced for the kernel, which generally increases as the situation becomes more register constrained. Without live register identification, the situation is deep into starvation for all register counts shown, and the quality of the resulting schedule is nearly identical in each case. With live register identification, a rise in total critical path can be seen as starvation gets worse. The step count increase (figure 4.52) lags the critical path increase, showing that the random rewind attempts often lead to the data paths in a step being made more combinatorial in preference to pushing them into a later step.

This example demonstrates that the register starvation avoidance techniques allow a valid schedule to be obtained on a core with 12 (i.e. $44 - 32$) fewer registers than can be achieved with just the scheduling algorithm on its own—i.e. a 27% improvement in schedulability in this case. It is believed that this may scale with the complexity of the basic block data flow graph. As of the time of writing, no example has been found where a valid schedule cannot be obtained using register starvation avoidance.

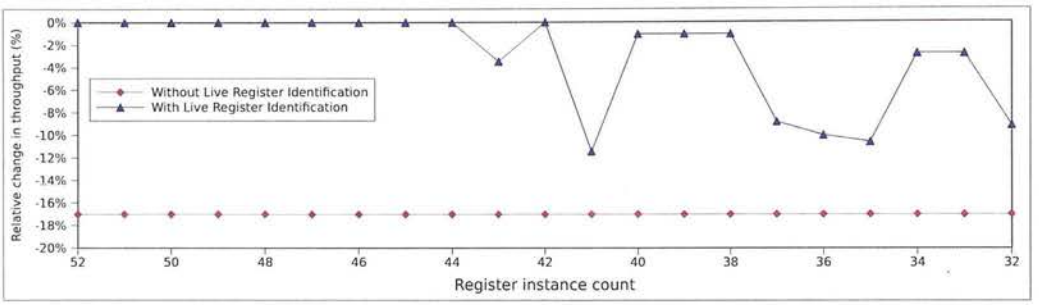


Figure 4.54: Change in throughput of the resource constrained gamma correction module, over a range of register instance counts. The throughput generally reduces as the situation becomes more register constrained, and is affected by a combination of the increase in step count and the increase in total critical path. In any case, the reduction in throughput is mild.

4.13.4 Results: Global Register Reallocation

A complete 3rd party image signal processing (ISP) pipe was used to investigate the effect of register renaming (introduced in section 4.12.2 on page 115) on routability and path lengths. The example ISP has a complexity of around 800 operations per pixel, and targets a RICA array of 1000 cells. The modules of the ISP are grouped into 3 kernels, which are executed one after another for each line of the image.

Mapping the connections of each step onto the real device requires the generation of paths along the interconnect resources. To reduce the path lengths, cells are reallocated to bring connected cells closer together. Cells which have internal state, such as registers, must be reallocated consistently across all steps in the program. The mapping tool achieves this by choosing a fixed allocation for these cells the first time they are used. This means that their locations are optimum only in the first step where they are used. To minimise the consequences of this, the steps are operated on in descending order of complexity[48], so that the steps that are more difficult to route are given the least restrictions. Typical steps contain many connections to and from registers, so their positions (allocation) have a large effect on the path lengths in the step overall.

		Interconnect utilisation		Average cx. delay (ns)	
Kernel	Connections	Disabled	Enabled	Disabled	Enabled
L1048 (first)	825	16.63%	16.36%	0.966	0.963
L917 (second)	710	21.92%	15.39%	1.55	1.10

Table 4.8: Post-routing statistics for the two most complex kernels in a 3rd party ISP pipe, with register renaming either disabled or enabled. L1048 will be freely reallocated in each case, but L917 will have some fixed allocations when register renaming is disabled.

[48] in terms of connection count.

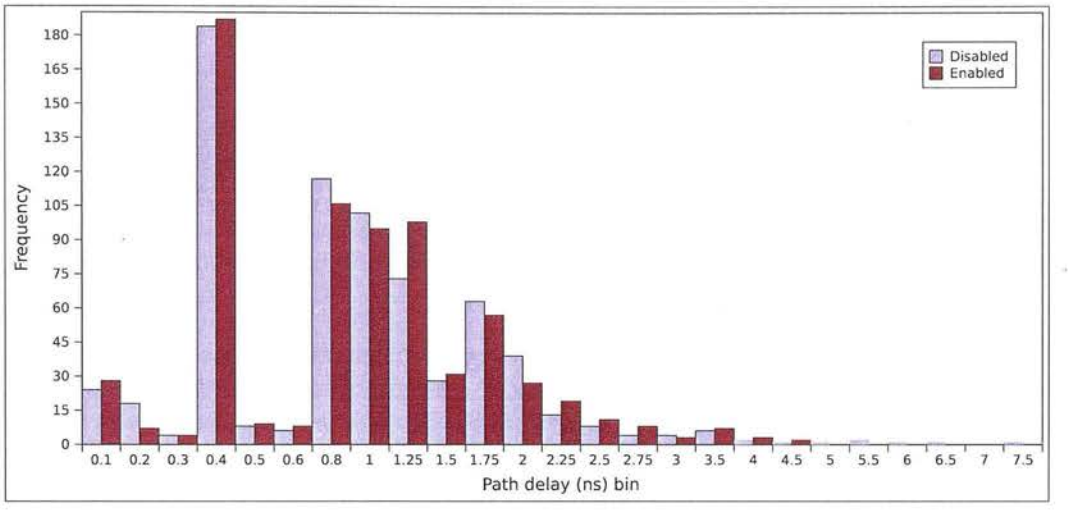


Figure 4.55: Histogram of path lengths for each connection in the L1048 kernel, with and without register renaming. Registers in this kernel are freely reallocated in both cases, so differences are the result of variation between runs. The bins are distributed logarithmically. Register renaming can be seen to make shorter paths more common, with no paths longer than 4.5ns (compared to a maximum at 7.5ns without renaming).

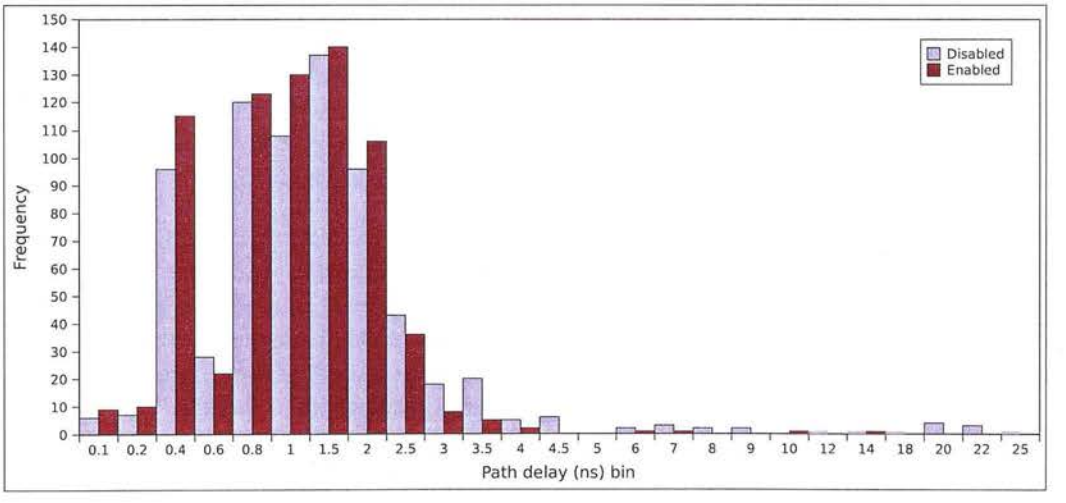


Figure 4.56: Histogram of path lengths for each connection in the L917 kernel, with and without register renaming. Some registers in this kernel have a fixed allocation when register renaming is disabled. The bins are distributed logarithmically. Register renaming can be seen to make shorter paths more common, with a single outlier at 14ns (compared to a maximum at 25ns and several in the 20–22ns range without renaming).

Register renaming uses the global register reallocation information (obtained using the algorithm described in section 4.12.1 on page 113) to make all registers freely reallocatable in each kernel. Without this information, only the first kernel to be operated on is able to freely reallocate registers. The example was chosen based on it being the available application with the

largest kernels (and thus the most variation in interconnect path length and placing the most pressure on the interconnect), with more than one kernel so as to demonstrate that optimisation is possible on each kernel individually (as opposed to performing a global reallocation tailored for one kernel, but potentially compromising the others).

Results were obtained using a mapping tool that is still under active development. Results are limited to the two most complex of the kernels. Excessive run times prevented sufficient runs to be performed to completely characterise the variance between runs. Therefore, these results should be considered to be merely indicative.

Table 4.8 shows pertinent connection statistics for this example, for two runs—with and without register renaming. Figure 4.55 and figure 4.56 give a more detailed view of the resulting connections for the two kernels, in the form of histograms. These show the number of connections that lie within each range (bin) of critical paths displayed on the x-axis. The bins are distributed logarithmically to show more detail in the lower lengths, where most of the connections lie.

As expected, the results show little variation between the two runs for the first kernel (L1048), where all registers are reallocatable. The relative difference in average path length is 0.31%, and the relative difference in interconnect usage is 1.6%. This means that most of the difference seen in subsequent steps should be attributable to the level of freedom to reallocate. The corresponding histogram (figure 4.55) shows very similar distributions, with very few connections lying outside of the bell. The maximum connection length is 7.4ns.

For the second kernel (L917), without register renaming, a significant number of registers are locked in a sub-optimal allocation. Register renaming should free these. The results show a 30% decrease in both average path length and interconnect utilisation. The connection length histogram (figure 4.56) consistently shows more connections lying in most of the lower bins, and a drastic reduction in the number of connections in all the higher bins.

If more kernels were processed, without register renaming, the number of registers locked in place will gradually increase as each successive kernel is processed. Therefore, the improvement with register renaming should become increasingly apparent. Also, it should be noted that this example was not pipelined. Enabling pipelining significantly increases the number of connections involving registers, and as a result, register renaming should affect a larger percentage of the connections, and thus have an even larger effect on the quality of the step.

4.14 Summary

This chapter looked at algorithms used in the process of converting a program described in a high-level language into configuration contexts for any arbitrary RICA core. The program is first compiled using a conventional industry-standard compiler (GCC) using a custom back-end which specifies an instruction set matching the capabilities of the individual cells present in the target array. The algorithms presented in this chapter are used to convert this assembly into configuration contexts, reconstructing the parallelism inherent in the basic blocks of the program, whilst adhering to the available resources.

The process of extracting parallelism from basic blocks involves inferring additional registers. A series of algorithms were introduced that allow register life times to be derived from the assembly, and to work around register starvation by gradually reducing the amount of parallelism until a valid schedule can be obtained. Register lifetime information was also used to perform register reallocation to improve the mapping onto the reconfigurable fabric, reducing the interconnect paths.

Data path machines are able to perform operations sequentially, in parallel, or combinatorially (i.e. operation chaining). The data paths produced by the compiler often consist of many operations chained together, which can lead to long critical paths after parallelisation. These limit the achievable throughput. The next chapter looks at ways to significantly improve upon this, by breaking these combinatorial chains into pipeline stages.

Chapter 5
Pipelining

Streaming applications such as real-time signal processing demand high throughputs, and are becoming increasingly prevalent in low-cost embedded systems, such as mobile phones. To meet the tough throughput and area requirements, ASICs have usually been the best solution on a performance/cost basis. However, as the cost of ASIC design and manufacture ever increases, and as customers demand more and more functionality, separate ASICs for each set of streaming algorithms—whether discrete or as part of a larger SoC—becomes a costly exercise. Many of these features are not used at the same time, so there is room for silicon re-use. Furthermore, vendors often wish to differentiate their products by providing a different set of algorithms for a particular feature compared to their competitors. A reprogrammable solution addresses these problems by decreasing the NREs of developing the initial ASIC (by sharing it amongst a wider audience), and allows different applications to use the same silicon at different times.

As discussed in earlier chapters, there are two main families of reprogrammable solutions that are able to meet these throughput requirements: SIMD architectures (such as modern GPUs), and reconfigurable data path architectures (such as embedded FPGAs). SIMD architectures achieve high throughputs by operating on several iterations of data at once (distributed across several execution units), making up for the lack of performance of each unit (a microprocessor). However, this is only possible on *embarrassingly parallel* [84] algorithms, where there are few data dependencies (i.e. finite impulse response (FIR) filters). Data path architectures operate on a smaller batch of data at once (often just a single iteration), but the latency of each iteration is significantly lower, thus achieving a higher throughput per unit. Operating on a smaller batch size means that data path architectures have a higher tolerance to infinite impulse response (IIR) filters.

Coarse-grained reconfigurable data path architectures, such as instruction cell based processors [5][6], are better suited to embedded systems than FPGAs, as the routing/computation area ratio is lower, and the smaller configuration sizes reduce the program memory footprint. The configuration size also allows these devices to be reconfigured much more rapidly, thus allowing them to implement control flow similar to a micrprocessor.

This chapter looks at ways to improve the throughput of streaming applications on coarse-grained reconfigurable architectures that support a high degree of operation chaining. Performance is optimised by attempting to match the size of each kernel—the inner loop where most of the execution time is spent—to the available resources, allowing them to fit into a single configuration. This allows the configuration to persist for many clock cycles, operating on new data on each cycle. This increases throughput, since no time is spent having to reconfigure the core between successive iterations. It also decreases power consumption, as the configuration only

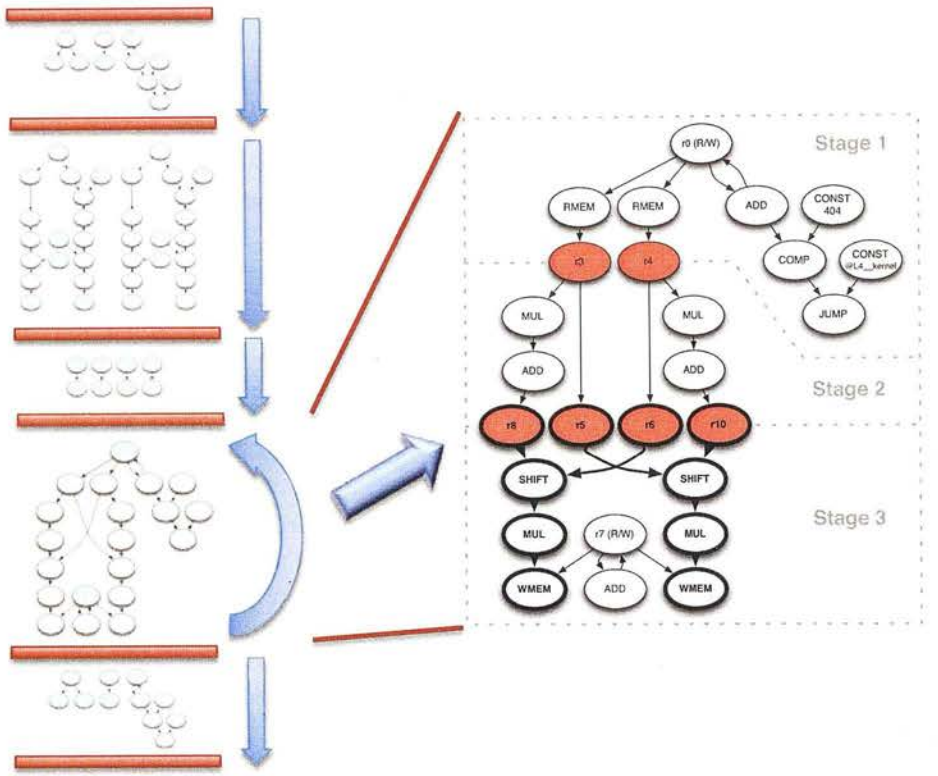


Figure 5.1: Typical program running on a dynamically reconfigurable processor—the program consists of several steps running in sequence, in which some can loop back to themselves (i.e. are kernels). The kernel can be pipelined, increasing its throughput.

needs to be fetched from program memory (or cache) once—upon first entering the kernel—rather than on every iteration. However, the resulting data paths can often have a long critical path, leading to poor temporal utilisation of the functional units, since they have to wait until all functional units have completed before operating on the next batch of data, which limits the throughput.

Pipelining provides a way of starting to operate on a new batch of data before an old one has completed. This allows the functional units of multiple stages of the kernel to be active concurrently; each operating on a different batch of data. The technique allows complete kernels that were mapped to a single configuration context, to have their critical path length decreased by the addition of pipeline stage registers, as illustrated in figure 5.1.

Section 5.3.1 describes an algorithm to perform pipeline stage allocation, based on a given target critical path constraint. Section 5.4 shows how properties of dynamic reconfiguration can be used to fill and flush the resulting pipeline. This is an entirely software approach. Section 5.5 proposes a second approach, which introduces some changes to the hardware, to allow filling and flushing to be incorporated into the single kernel configuration context, thus significantly reducing the program memory overhead—especially for very deep pipelines. Section 5.6 details how the task can be completely automated, by first suggesting how to identify loops that can

be pipelined, and second by introducing an algorithm for finding the optimal target critical path constraint. Section 5.7 shows how further improvements can be made by internally pipelining the hardware of the instruction cells, thus reducing the minimum possible pipelined critical path. Section 5.8 shows the result of applying these techniques to a real-life kernel used in image processing.

5.0.0.1 Aims

- Automatically pipeline compute-intensive loops to significantly increase throughput.

5.0.0.2 Objectives

- Automatic pipeline stage assignment, based on a user-supplied target critical path constraint.
- Minimal hardware changes.
- Minimising the impact of pipelining on the context configuration size.
- Minimising the impact of pipelining on the overall program size.
- Automating the choice of target critical path.

5.0.0.3 Novelty

- Combining ASIC design techniques (structural pipelining) and rapid dynamic reconfiguration to achieve automatic pipelining of kernels, in a manner similar to software pipelining. *Dynamic pipelining* (section 5.3), using a *Pipeline stage allocation algorithm* (section 5.3.1), and *Automating the choice of timing constraint* (section 5.6).
- A method for removing the need for separate pipeline fill and flush contexts with minimal addition of hardware—*Single-step pipelining* (section 5.5).
- *Support for pipelines involving internally pipelined cells* (section 5.7).

Another way to look at the first item is that the structural pipelining allows a custom execution unit pipeline to be obtained, that best matches the algorithm. Then conventional software pipelining is used to partition the software—the instructions of that kernel— to fit that custom pipeline. The second item is then how to implement hardware predication in a manner that allows a single configuration context to perform all the phases of a software pipeline: fill, loop, and flush. The last item is useful for further improving the throughput by hiding long combinatorial operations or memory access latency.

5.1 Background: Structural Pipelining

Various approaches of pipelining data paths have been proposed [85, 86]. These require that the designer specifies a throughput constraint, in order to allow the algorithm to best make the choice between throughput and the area overhead each pipeline stage introduces. These approaches describe various algorithms for the task of pipeline stage allocation, applied to a number of different levels in a design, from high-level modules described at the behavioural level, down to the operation-level [87]. This is possible due to the hierarchical nature of designs described via hardware description languages (HDLs). In the context of computing, pipelining can be applied at the structural-level, where data paths are defined between abstract building blocks—which map to the functional units. In addition to constraints imposed by the designer, [85] describes how the presence of feedback loops in a design limit the extent to which pipelining is possible. An important part of the optimisation process is therefore to minimise the presence of feedback loops. Techniques such as retiming [88] may be used for this purpose.

In an ASIC environment, pipelining is often done together with component selection, in order to make the additional trade-off between the cost of higher performance components, and area for equivalents made from pipelining lower-cost components [89]. This can also be seen in an FPGA environment [90, 91], where the choice can be made between scarce special-purpose resources or synthesising the functionality using configurable logic blocks, pipelined to achieve sufficient throughput. In a computing environment, component selection could be viewed as the choice of which instruction expansions and manipulations the compiler should perform in order to best match the available functional unit resources, or in resource selection for custom cores. The key difference in computing environments compared to ASIC/FPGA environments is in the level of re-use of functional units, and the time scale over which they are re-used.

For architectures that support instruction chaining, scheduling involves mapping as many dependent and independent data paths into as few configuration contexts as possible [57]. Independent data paths run in parallel, so the time for which a configuration persists is determined by the maximum critical path length of these data paths. If sufficient functional unit resources are available, loops can be optimised by loop unrolling [92]—i.e. placing multiple iterations as independent data paths in the same configuration. This allows multiple iterations to begin and end at once. This does not change the original critical path length, yet can increase the throughput.

The throughput is determined by the critical path length of a loop iteration and the number of iterations that can be performed at once. During each execution of the loop configuration context, data propagates through the operation chains until the final result is ready. This means that the functional units involved in that chain are only performing useful work for a fraction of the time. This is where loop pipelining techniques [7, 8, 9] come in, where the data paths are structural-level pipelined—to artificially reduce the critical path length by allowing new iterations to begin without waiting for the completion of previous iterations. This can be thought of as successive iterations of the loop being replicated in hardware, but offset from each other to deal with the data dependencies between the iterations.

The same technique can be applied to SIMD architectures: the algorithm can be split into pipeline stages, and each stage mapped to a different execution unit. This reduces the execution time per iteration for each unit, which allows infinite impulse response filters to be accelerated.[1] This also reduces the code size for each unit, which can be useful in fitting more complex algorithms into the available resources of each unit. This is used in large finite impulse response filters, where the pipeline can be replicated—each operating on a different independent batch of data. This increases the throughput up to similar levels as having each unit perform the complete operation on a separate batch of data, but requires less code per execution unit.

5.1.1 Background: Software Pipelining

With increases in the number of functional units available in microprocessors and DSPs, techniques for maximising the utilisation of these function units have been devised, and are referred to as software pipelining [93, 32]. This involves rescheduling the instructions in a loop iteration such that multiple iterations are in progress concurrently—each at a different level of completion. This is done in such a way as to minimise the *initiation interval*—i.e. the rate at which new iterations are begun, which directly determines the throughput. If the available functional unit resources exceed the requirements of the operations in the loop, the loop may be unrolled. Each unrolled iteration uses a different set of registers, which prevents certain dependencies from conflicting with previous in-flight iterations. This allows the initiation interval to be further reduced, and hence increases the throughput. Additional instructions must be added before and after the software pipelined loop, in order to fill and to flush the pipeline. These are called the *prologue* and *epilogue*, respectively. This is illustrated in figure 5.2.

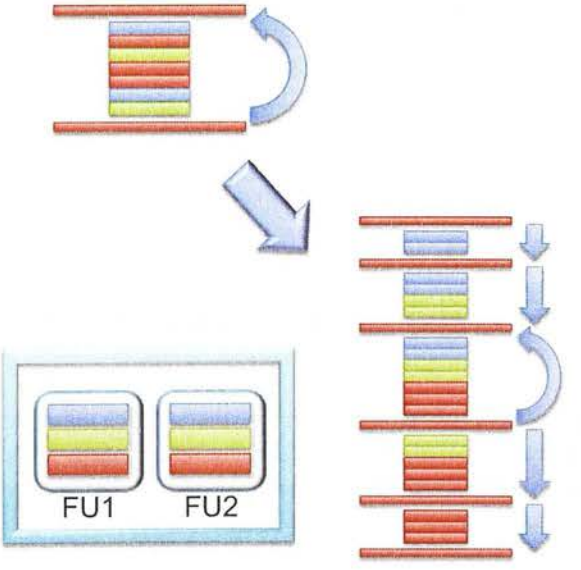


Figure 5.2: Software pipelining: simple loop running on a processor with two functional units that can run in parallel, but are fed from a single instruction queue. The colours show which instructions could be executed together without causing a pipeline stall (bubble).

[1] albeit only by the number of pipeline stages.

However, there is a limit to the depth of fixed pipelines that can be created without compromising the performance over a wide range of different applications [94]. There is also a practical limit on the degree of independent instruction-level parallelism that can be made use of [95], which places a limit on the number of functional units that can be made available. This practical limit can be avoided by allowing sequences of dependent operations to be chained together and executed in a single cycle—i.e. dependent instruction-level parallelism. Such sequences are statistically more common than there being sufficient independent data paths in a kernel to make full use of a large number of independent functional units. This is an approach that is being taken in some modern VLIWs/ULIWs [96, 31] and in the design of processors found in highly multi-core fabrics [1].

The remainder of this chapter describes how dynamic reconfiguration and operation chaining can be used to create custom hardware pipelines, that best match a particular kernel's requirements. Software pipelining techniques are then used to best map the kernel to this custom pipeline. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at run-time. This has the same effect as adding pipelining in hardware, but can be changed at run-time.

5.2 Preconditions

An operation is something that creates a value in the data path. Operations map to physical functional units in the core, or registers. Operations create values that are transferred through the routing network to other functional units in the core. Receivers of these values are either other operations, or global output registers—i.e. registers assigned by the compiler to make a value available in a later basic block. Reading from a global input register—i.e. a register assigned by the compiler to bring in a value from a previous basic block—is also counted as an operation, although writing to a global output register is not. Instead, each operation specifies which global output registers store its value (if any). This mirrors the information captured in normal assembly notation.

This work assumes that the target architecture exhibits the following properties:

- Sufficient functional units exist in the silicon to allow a kernel to be mapped into a single configuration.
- Operations can be chained together directly through the interconnect network, or via registers. For simplicity, it is assumed that any direct connection can be replaced by a register; although this restriction could be easily overcome.
- Registers introduce a delay of one iteration into the data path—i.e. any value written to a register first becomes available in the next cycle.
- Arbitrary program flow control (branching) is supported, such that if the value of the program counter is modified, the configuration at that address is loaded on the next cycle. The mechanism for modifying the program counter is referred to as the *jump* operation in this paper.

It is assumed that the kernel is intended to run for many iterations. Since the additional configuration contexts introduced with pipelining—i.e. the prologue and epilogue (introduced in section 5.4)—have to be loaded and executed once each, irrespective of the number of required iterations of the kernel, the total time spent configuring the core is likely to be higher when pipelining is used. Therefore, the iteration count of the kernel must be high enough so that this increase in configuration time is offset by the decrease in total execution time for the same number of iterations of the kernel, resulting from the decrease in critical path length in the kernel loop context. Furthermore, it is impossible for the pipelined design to perform fewer iterations than the number of pipeline stages present. The minimum control flow path would be to execute each of the prologue contexts in sequence, then the kernel loop context once, then each of the epilogue contexts in sequence—during which there are as many in-flight kernel iterations as there are pipeline stages (see figure 5.5).

The tool chain in the current implementation consists of a compiler that produces assembly consisting of operations that map to the functionality of cells in the core, and a scheduler that extracts the parallelism from the basic blocks of the assembly, creating the netlist that defines each of the configuration contexts that can be loaded onto the target architecture. Pipelining is performed on the netlist. Since this is done outside of the compiler, much of the information available as part of the compiler's data model is no longer available by this stage. Therefore, much of the configuration cannot sensibly be modified without unexpected adverse effects. This limitation, although avoidable, aids in making the proposed technique more general.

For use as part of a re-targetable tool chain, it is desirable to have as few in-built assumptions or special case logic as possible. The special case logic used in this work has been reduced to just the following:

In addition to the dependencies implied by the connectivity between the function units relating to the operations in the kernel, it is also necessary to capture other dependencies implied by the original order of execution. These are dealt with by adding constraints, to preserve the original temporal order. Examples include *volatile* operations—i.e. operations of a type that requires the execution order of operations of that same type to be preserved—and potentially aliasing read/write operations on the data memory.

The execution count of each operation remains unchanged after pipelining. This is a simple way to ensure that side effects (e.g. state changes) of each operation remain unchanged, without having to explicitly define which operations can have side effects. There are a few cases where particular side effects require special treatment (e.g. the jump operation and registers).

The jump operation has an immediate side effect of causing the kernel to exit, and passes execution onto the pipeline flushing contexts (epilogue—introduced in section 5.4). Therefore, the jump operation must be identified, and placed in the *first* pipeline stage—for reasons described in section 5.4.

An important side effect of registers transferring values from one cycle (and thus iteration) to the next, is that any chain of operations that reads from a register, then writes back to the same register (i.e. a feedback loop), must be placed in the same pipeline stage. Otherwise, it would take more than one iteration to update the value of the register, thus causing the pipeline to operate on garbage for some iterations. A feedback loop is also possible involving access to data memory, which also introduces a delay of one cycle. However, use of data memory for feedback is discouraged, since the latency cannot be hidden.

Since the critical path of the pipeline as a whole is dictated by the maximum of the pipeline stage delays, and feedback chains (and the jump chain) must be placed into a single pipeline stage, the length of such chains dictates the minimum possible overall critical path length—and thus dictates the maximum possible throughput. Therefore, the compiler should ideally perform optimisations that minimise the length of the feedback chains. The description of the target architecture must include cost weighting for each functional unit, so that the compiler can calculate the critical paths.

The user specifies the desired throughput target for the kernel basic block, by specifying the desired critical path length of the kernel loop. This is done via special mark-up in the assembly. The pipeline stage allocation algorithm uses this to determine where to insert pipeline stage registers, and thus indirectly determines the number of pipeline stages to generate. Pipelining is only applied at the request of the user, via the presence of the aforementioned mark-up, since the iteration count is unknown in the current data model. For reasons of generality, no attempt was made to add code to determine the iteration count, although this is a possible future improvement.

5.3 Contribution: Dynamic Pipelining

Conventional structural-level pipelining can be applied to single configuration context kernels with long critical data paths, in order to reduce the critical path, and thus increase throughput. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at runtime. This is done using existing register resources in the core to delay values for a single execution cycle, allowing values to be bridged across pipeline stage boundaries.

Structural pipelining is applied to the kernel basic block by first assigning each operation in the original data flow graph to a pipeline stage. Then, registers are introduced to store values over boundaries between pipeline stages. Only those values that are used in later pipeline stages are stored. A new register is needed for each value for each pipeline stage boundary over which it must persist. Figure 5.3 shows an example kernel before and after structural-level pipelining. The example includes only simple feedback chains consisting of a simple increment of the value of a register, however more complex feedback chains are also possible.

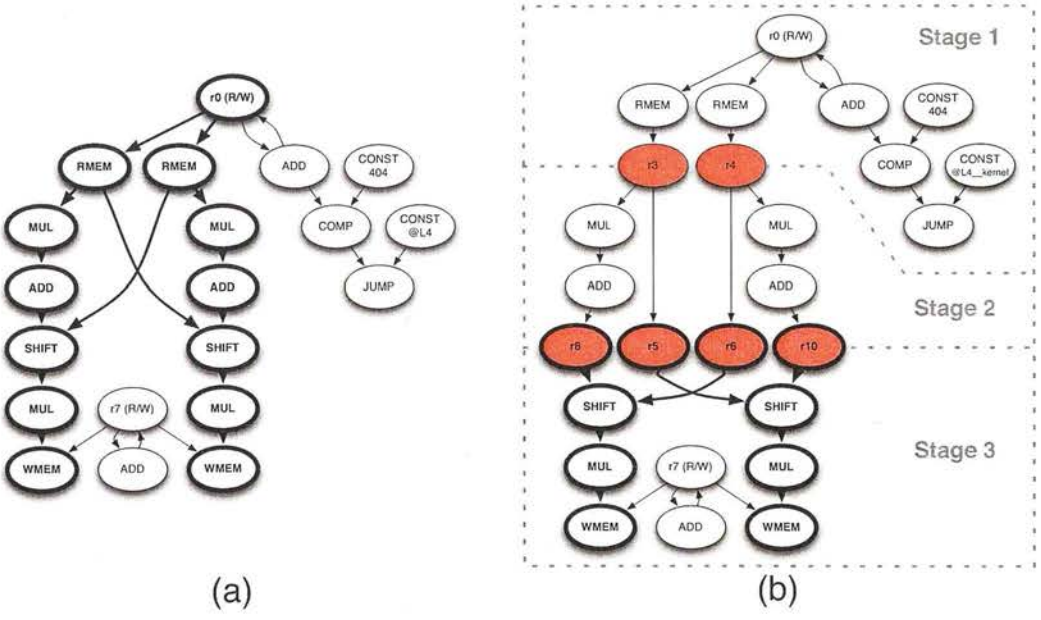


Figure 5.3: Example kernel data flow graph, (a) before pipelining, (b) after pipelining (kernel loop context). The inserted pipeline stage registers are shown in red. The per-cycle critical path is shown in bold, and is shorter in (b), which allows for a higher throughput.

5.3.1 Contribution: Pipeline Stage Allocation Algorithm

First, constraints are defined between operations, where the order of execution is important. Examples include *same stage or earlier* constraints between operations reading from input registers and operations that have those same registers marked as global output registers, and *same stage or earlier* constraints between data memory read operations and potentially aliasing data memory write operations. All operations in a feedback chain must be placed in the same pipeline stage, since such chains require single-step total latency in order to keep the pipeline full. The algorithm for assigning pipeline stages to each operation is as follows:

- Identify the *jump* operation, and all of its dependencies. Save this in a set—the *jump chain set*.
- Create the *remaining set*—a record of those operations yet to be assigned to a pipeline stage. This is initially populated with all the operations except for those in the *jump chain set*.
- Define the constraints:
  - Add *same stage or earlier* constraints between operations reading from input registers, and operations that have those same registers marked as global output registers.
  - Add *same stage or earlier* constraints between data memory read operations and potentially aliasing data memory write operations.
  - Add *same stage or earlier* constraints between volatile operations of the same kind, to ensure that they still appear in their original order.
- Detect feedback chains:
  - Identify all the operations that are part of each feedback chain, and record them in a set for each chain. These shall be referred to as the *feedback sets*. No operation in a feedback set may be assigned to a pipeline stage until all the operations in that set are ready to be assigned.
- Create an ordered list of pipeline stages, initially consisting of a single entry. Each entry contains the set of operations that have been assigned to that pipeline stage.
- For each operation in the *remaining set*:
  - Create a temporary set containing this operation and any operations in the same *feedback set* (if one exists).
  - Determine whether any of the operations in the temporary set have any successors that are also in the *remaining set*. If they do, then the temporary set is not ready, so discard it and move on to the next operation in the *remaining set*.
  - Determine whether any constraints involving the operations in the temporary set involve operations that are also in the *remaining set*. If they do, then the temporary set is not ready, so discard it and move on to the next operation in the *remaining set*.
  - Identify the latest pipeline stage where all the operations in the temporary set could be placed, according to their dependencies and constraints.

  - Construct a configuration context containing all the pipeline stages constructed thus far, and calculate its critical path delay[2].
  - Speculatively construct a configuration context containing all the pipeline stages constructed thus far, including the operations from the temporary set, placed in the previously identified pipeline stage. Calculate its critical path delay.
  - If the critical path delay is different (i.e. increased), and the new delay exceeds the target, then move to the preceding pipeline stage[3]. Long connections may have delays many times greater than the target, in which case the insertion point is moved back by several pipeline stages.
  - Transfer the operations from the temporary set to the identified pipeline stage, and remove them from the *remaining set*.
  - Loop whilst the *remaining set* is not empty.

- Add the operations from the *jump chain set* to the first pipeline stage.

The algorithm is a form of list scheduling. Only operations whose predecessors (in the data path) have already been assigned a pipeline stage may be considered for insertion on each pass. In order to minimise the register count, operations should be placed in as late a pipeline stage as possible. Operations that must be placed in the same stage are dealt with together. Operations are considered for placement in the earliest pipeline stage containing any of their successors. Then, the insertion point is moved towards earlier pipeline stages until all constraints have been satisfied. Once a valid insertion point has been identified, the critical path is calculated for the resulting (incomplete) configuration context with the operation in that pipeline stage. If the critical path meets the target value, the operation is placed in that pipeline stage. Otherwise, the operation is added to an earlier pipeline stage—the gap (number of stages earlier) being equal to the critical path divided by the target critical path. This allows for the interconnect itself to be pipelined, where pipeline stages can contain just wires between pipeline stage registers.

The creation of dependencies ensures that the sequence of state changes is maintained, thus ensuring correct results. Assigning operations to a late a pipeline stage as possible aids to reduce the number of registers required. Once the pipeline stages have been determined, pipeline stage registers are assigned as follows:

- For each pipeline stage in sequence:
  - Assign a new register storing the value produced by each operation in all previous pipeline stages that needs to be stored for use in this or any later stage.

[2] including the reading from and writing to pipeline registers.

[3] creating a new pipeline stage at the beginning of the list, if the chosen stage was the first in the list.

5.3.1.1 Limitations

The pipeline stage assignment algorithm described here cannot pipeline data flow graphs containing large-scale feedback loops[4] involving registers in the core, where the final result of one iteration is involved in the calculation some iterations later. The data flow graph would show the final result being fed through a chain of registers, back into beginning of the graph. The register feedback chain detection logic would see this entire chain as all having to be in the same pipeline stage. In reality, they do not really need to be in the same stage; the feedback registers could instead be re-used as pipeline stage registers. This re-use of existing registers as pipeline stage registers is called *re-timing*, and is outside the scope of this thesis.

[4] i.e. infinite impulse response filters.

5.4 Contribution: Multi-Step Pipelining

Normally, a pipelined design would require additional logic to take care of initialising the pipeline stages, or to suppress the operations in later pipeline stages until the previous stages have filled (predication), so that they do not operate on garbage. However, the pipelines in a coarse-grained DRA are themselves rendered as part of the configuration context. Provided that the configuration time is not significantly larger than the execution time of each step, dynamic reconfiguration can be used to render different configurations before the main kernel loop configuration, to fill successive stages of the pipeline, and similarly to flush the pipeline after exiting the kernel loop. This allows the kernel loop configuration to assume that the pipeline stages are always full. This provides a generic, purely software alternative to predication, which can be used as a fall-back when no hardware support exists.

Fill: New configuration contexts are created to initially fill each successive stage of the pipeline. For n pipeline stages, $n-1$ pipeline filling contexts are created.

Loop: A single configuration context is created for the kernel loop, which includes all pipeline stages.

Flush: New configuration contexts are created to flush successive stages of the pipeline. For n pipeline stages, $n-1$ pipeline flushing contexts are created.

The core is dynamically reconfigured to first perform pipeline initialisation, then reconfigured to execute the kernel loop, then finally reconfigured to flush the pipeline—as demonstrated in figure 5.5. This is similar to the epilogue and prologue in software pipelining [32].

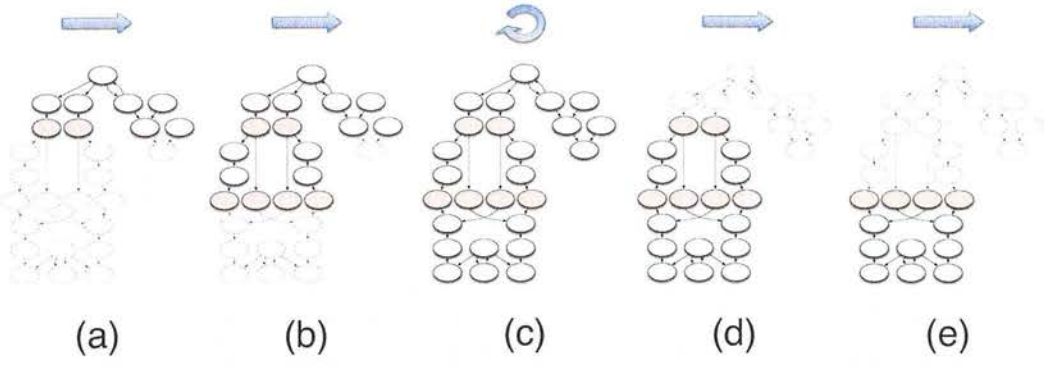


Figure 5.4: The sequence of configuration contexts created for the example kernel, (a) iteration 1—filling pipeline stage 1, (b) iteration 2—filling pipeline stages 1 and 2, (c) iterations 3 to $n-2$—pipeline full (loop), (d) iteration $n-1$—flushing pipeline stage 1, (e) iteration n—flushing pipeline stage 2.

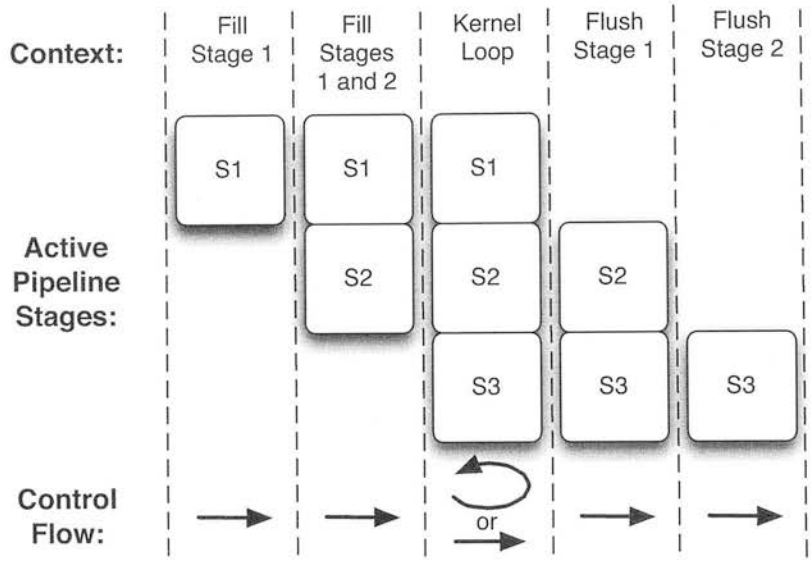


Figure 5.5: Control flow for a 3-stage pipelined kernel, showing which stages are active in each context (and moment in time). Execution flows from one context to the next, except in the kernel loop, which loops back to itself—holding the same context—until the end condition is satisfied.

The configuration contexts generated for the kernel example from figure 5.3 is shown in figure 5.4. The use of separate special-purpose configurations alleviates the need for special logic for this purpose in the kernel loop configuration context. Furthermore, there are no hardware-enforced limitations to which pipeline stage each operation can be placed in,[5], thus maximising the achievable pipeline depth.

Figure 5.5 shows which stages of the pipeline are active during execution for a 3-stage pipeline. As the target architectures may not be state free (e.g. memory access), it is important to not allow any operation in any pipeline stage to operate on garbage, and to preserve the execution count. With the arrangement shown in the figure, all pipeline stages will be executed the same number of times irrespective of the number of iterations performed in the kernel loop.

Now consider the original kernel, where the *jump* operation causes the loop to terminate after n iterations. In the pipelined kernel, we must ensure that the kernel loop terminates after n executions of the operations that calculate the loop termination condition; otherwise, the operations or operands would need to be modified to yield a different iteration count. Looking at figure 5.5, the minimum number of iterations possible in the pipelined design occurs when the kernel loop context executes only once. This corresponds to an iteration count equal to the number of pipeline stages (in this case 3). In order for the loop to terminate immediately, the operations that determine the loop termination condition must have been executed this number of times by the time the kernel loop context has been executed. This can only be achieved by placing these operations in the *first* pipeline stage. The same argument also applies for any higher iteration count. Figure 5.6 shows two examples, to highlight this point.

[5] only the data dependencies restrict this.

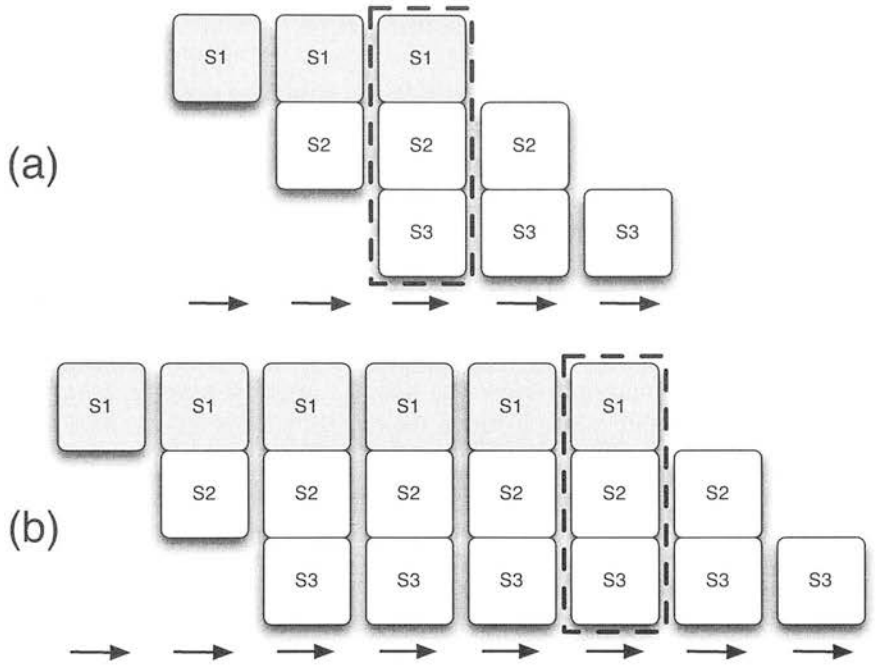


Figure 5.6: Expanded control flow for the pipeline shown in figure 5.5 for (a) 3, and (b) 6 iterations. The point at which the loop termination condition should evaluate to true is shown by a dotted box. It can be seen in both cases that only the first stage has executed for the desired number of iterations by this point.

Placing the *jump* in the first pipeline stage therefore requires that all of its dependencies are also placed in the first pipeline stage. Since the pipeline filling contexts (prologue) should always be executed in sequence (with no branching), the *jump* operation is omitted from these contexts, even though it is in a pipeline stage active in those contexts. Its dependencies are left in place, since their side effects are important—e.g. they could update the iteration counter whose value is used to determine the loop termination condition.

5.4.0.2 Limitations

Although the creation of separate sequences of fill and flush configuration contexts makes for a closer match to conventional software pipelining, giving maximum flexibility of which pipeline stage to assign each operation to, it introduces a significant program memory overhead. For example, in typical streaming applications where the programs consist of a few large kernels with a few steps of glue code in-between, pipelining the kernels can easily increase the step count (and thus program size) by an order of magnitude. However, since each of these contexts is a subset of the kernel loop context, they are good candidates for code compression. Even with compression, there will be a trade-off between the throughput of the loop context and the time taken to load each fill and flush context, in order to achieve the shortest total execution time.

5.5 Contribution: Single-Step Pipelining

For very large cores, data paths in kernels with high cell utilisation can have very long critical paths. It is these kernels that benefit most from pipelining, in terms of throughput. However, when using the dynamic pipelining technique with multiple configuration contexts, described in section 5.4, the overhead on the program memory of storing the series of fill and flush configuration contexts can be extremely prohibitive.

The sequence of fill and flush contexts each contain a subset of the pipeline stages in the kernel loop context. In other words, each of these contexts contain a subset of the active cells and connections in the kernel loop context, and no new information. As a result, they would seem to be good candidates for compression. However, since the operations involved in each pipeline stage can be arbitrary, it is difficult to partition the configuration stream in such a way as to make the partitions align with the pipeline stages of any given set of kernels. This means that real-time, hardware-based compression schemes may not be applicable.

This section explores modifications to the hardware that allow a kernel to be pipelined to an arbitrary depth, whilst only requiring a single configuration context (or at most, two configuration contexts—as will be explained in section 5.5.2) to be stored in memory.

At the most basic level, the idea is to execute the same configuration context (the kernel loop) for all three phases: *fill*, *loop*, and *flush*. As shown in figure 5.6, in order to maintain the original behaviour, each pipeline stage must be executed the same number of times as the original loop was to be executed. However, since each pipeline stage depends on the data produced by the previous pipeline stage in the previous iteration, each pipeline stage must be delayed by one iteration from the previous one. If one configuration context is to represent all of this, then this context must perform more iterations than the original loop. In fact, it must perform $n - 1$ additional iterations (where n is the number of pipeline stages). In these additional iterations, the operations of one or more pipeline stages will be operating on garbage.

To avoid the operations of each pipeline stage from being executed too many times, a mechanism is needed to disable them (*predication*). The next four sections present different ideas that gradually build up to a practical solution to this problem.

5.5.0.3 Idea 1

One simple way to prevent the cells from being executed too many times, would be to associate a hardware iteration counter with each cell, and initialise it (via the configuration) with a value stating how many initial iterations during the pipeline fill phase it should be disabled for. This also represents the number of initial iterations during the pipeline flush phase that it should be enabled for (after which it would become disabled).

However, this approach introduces a substantial overhead in the configuration size, since each cell would have an initial counter value associated with it.

Even if this was stored separately to the rest of the configuration stream, so as to only impose these additional configuration bits to pipelined kernels, it is still a significant overhead. Furthermore, the number of bits in each of these fields imposes a maximum possible pipeline depth of $2^n - 1$. Since the configuration size overhead scales linearly with this bit width, there would be significant pressure to make this value as small as possible.

5.5.0.4 Idea 2

The first observation is that only operations that have internal state can affect the overall behaviour of the program if they operate on garbage. N.B. purely combinatorial operations that are executed more times than necessary (operating on garbage in these additional iterations) will have no effect on the state of the system, and thus have no effect on program behaviour; they merely introduce a small increase in power consumption. Therefore, it is only necessary to control the iteration count of cells that maintain internal state.

The cells that have internal state generally come under two categories: *registers* and *hardware I/O*. Hardware I/O cells act as interfaces to external hardware, and these are few in number in a typical core. On the other hand, register cells represent a significant fraction of the cells in a typical core—especially a large core for use with pipelining. As a result, this first observation can be used to reduce this configuration overhead by maybe 50% for a typical core, by only providing initial values for the iteration counters of cells that maintain internal state.

5.5.0.5 Idea 3

The effect on a register cell of operating on garbage is to replace its internal value with garbage. Therefore, disabling a register cell simply prevents it from replacing its internal value with the value currently read from its input port. The astute reader may correctly point out that the majority of registers in a typical large core are used as pipeline stage registers—to bridge values from one pipeline stage to the next. Having a large number of them increases their chance of being close to where they are needed in the pipelined data paths, thus decreasing the critical paths.

In the case of pipeline registers, the initial value stored in the register is of no consequence during iterations when that pipeline stage is still awaiting data during pipeline filling, and similarly, the final value stored in a register is of no consequence after that stage has completed the correct number of iterations.

So, one solution would be to introduce a separate class of register cell specifically for pipelining. Such cells would not have a counter associated with them. However, this poses a significant problem: we cannot sensibly choose in advance which registers should be expressly for pipelining, and which should be normal registers. Any mismatch between the shape of the pipelined data paths and the core would result in additional path lengths, which may severely impact performance, and require more pipeline stages to be created, making the problem worse. This is the same problem as why compression is difficult.

5.5.0.6 Idea 4—A Practical Solution

The second observation is that out of the cells that have internal state, hardware I/O cells are normally *disjoint*—their output does not depend on their input in the current iteration. In fact, input to the cell and output from the cell tend to (by design) be distinct, independent data streams within that kernel. The result of this is that the output side of an I/O cell tends to act as a data source bringing data into the kernel, and thus appears in a very early pipeline stage; and the input side of an I/O cell tends to act as a data sink bringing data out of the kernel, and thus appears in a very late pipeline stage. Generally, moving the input side of all hardware I/O cells active in the kernel into the first pipeline stage, and moving the output side of all hardware I/O cells active in the kernel into the last pipeline stage, simply introduces a slight increase in the number of pipeline stage registers needed.

As a result, we can (in effect) hardwire the counter associated with the output side of the hardware I/O cells to 0, and the input side to $n - 1$. This obviates the need to store the initial value in the configuration stream.

A third observation is that registers that are both read from and written to in the kernel (termed *kernel registers*) mostly act as counters, and thus appear in early pipeline stages. Typically, their stored value is used as a data source to other operations further down the critical path of the kernel, and their new value is simply the current value incremented (or some similar operation with a short critical path). We can therefore move most of these kernel registers into the first pipeline stage, without affecting the ability to pipeline.

With all registers appearing in the first pipeline stage, the counter value associated with them will always be initialised to 0, and as a result, there is again no need to store it in the configuration stream. This completely eliminates the need to store additional data in the configuration stream.

However, there is a problem: some kernel registers are not used as simple counters, and instead are written to from further down the critical path of the kernel. This often occurs when partial results from previous iterations are to be re-used, instead of re-calculated. Forcing all such registers into the first pipeline stage severely limits the extent to which the kernel can be pipelined—often preventing it from being pipelined at all.

A further observation is that in almost all cases identified, these kernel registers do not bring data into the kernel from previous steps. Therefore, the initial value that they store is not important, and can be safely overwritten with garbage during pipeline stage filling, without affecting program behaviour. As a result, these can be safely placed in any pipeline stage.

In the very few cases where the initial value of kernel registers is important, some additional instruction cells can be used to preserve the initial value, as will be described in section 5.5.2.

5.5.1 Contribution: Hardware Modifications For Single-Step Pipelining

Recall that one of the distinguishing features of the target architecture is that it is in control of its own reconfiguration—i.e. the state of the machine and its data paths determine which configuration context to load and when. The `jump` cell is responsible for reading the next jump target from the data paths of the current configuration context, and the condition for when this target should be loaded into the program counter.

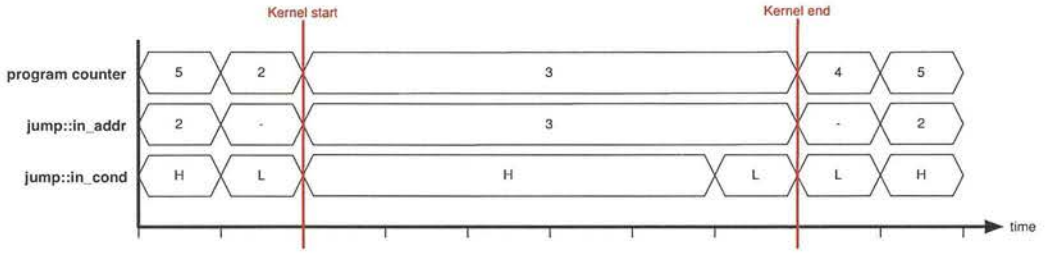


Figure 5.7: Internal control signals during execution of a normal (non-pipelined) kernel (step index 3), within an outer loop (step indexes 2, 3, 4, and 5). `jump::in_addr` and `jump::in_cond` are read from the reconfigurable data paths. The program counter is internal to the jump cell, and shows the index of the configuration context currently loaded in the core. The values shown for the data paths are those once they stabilise near the end of the current iteration.

When executing a kernel (i.e. a configuration that loops to itself), the jump target is the kernel itself. The data paths and state are constructed such that the condition causes the kernel configuration context to loop back to itself until the appropriate iteration count has been completed, after which control passes to the next step in sequence. Figure 5.7 shows the state of the signals involved during this. In this example, a kernel (step index 3) executes for 6 iterations, inside each iteration of the outer loop (step indices 2, 3, 4, 5). During execution of the 6th iteration of the kernel, the data paths driving the `in_cond` input of the jump cell stabilise to zero, indicating that the jump back to the kernel should not be performed, and as a result control passes to the next step (index 4).

In order to support single-step pipelining, the main addition to this is the *pipeline depth counter* (PDC). This hardware counter is initialised to a value given in the configuration, equal to the number of pipeline stages in the pipelined kernel. The purpose of this counter is as follows:

- To delay switching to the next configuration context, to allow pipeline flushing to take place.
- To disable during pipeline filling, those cells with internal state that must be in the last pipeline stage.
- To disable during pipeline flushing, those cells with internal state that must be in the first pipeline stage.

The jump cell is further extended to contain a new 2-bit state representing the current *execution phase*, which can be one of the following:

Normal: Normal (non-pipelined) step execution.

Filling: Filling the pipeline stages of a pipelined kernel.

Looping: Executing the loop of a pipelined kernel.

Flushing: Flushing the pipeline stages of a pipelined kernel.

Upon entering the pipelined kernel,[6] the execution phase is set to *filling*, and the PDC is initialised to the number of pipeline stages minus one. The PDC is decremented by one after each iteration of the kernel, until it reaches zero. When the PDC reaches zero, the execution phase shifts to *looping*. When the jump cell's inputs signal the end of the kernel, the execution phase shifts to *flushing*, and the PDC is set to the pipeline depth minus one again. The current configuration context continues to execute (i.e. the jump is delayed) whilst in the *flushing* phase. The PDC is again decremented after each iteration, and upon reaching zero, the execution phase shifts to *normal*, and the next configuration in sequence is executed.

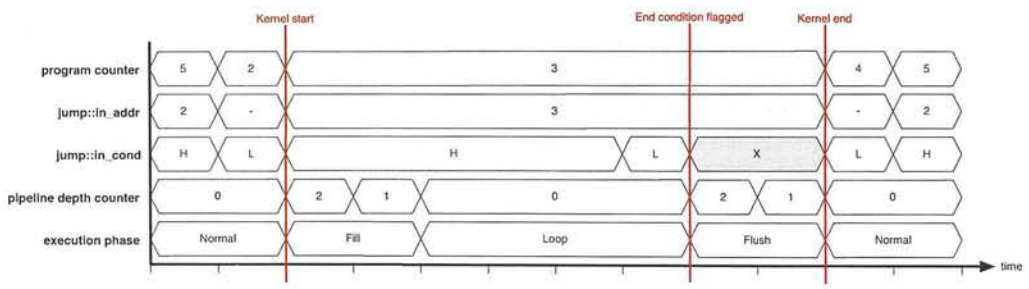


Figure 5.8: Internal control signals during execution of the kernel from figure 5.7, this time (single-step) pipelined into 3 stages. The two new signals are also internal to the jump cell.

This mechanism increases the iteration count for the kernel by $n - 1$ (where n is the number of pipeline stages), and produces signals to indicate when filling and flushing. Figure 5.8 shows the same kernel and outer loop as in figure 5.7, but the kernel has been pipelined into 3 stages. The figure shows the normal execution outside of the kernel (the *Normal* execution phase), and the sequence of *Fill*, *Loop*, and *Flush* inside the kernel. After six iterations—the desired duration of the inner loop—the data paths driving the jump cell's inputs indicate that it is time for the kernel to end, so the jump condition goes low. However, this time another two iterations of the kernel are performed, to allow the pipeline stages to be flushed, before allowing control to pass to the next step (index 4).

The cells which maintain internal state, which must either be in the first or last pipeline stage, can monitor the current execution phase to determine when they should be disabled. Note that many cells are *disjoint*, where the input and output sides are independent. In these cases, the input side is in the last pipeline stage so disables itself during the filling execution phase, whereas the output side is in the first pipeline stage and so disables itself during the flushing execution phase.

[6] indicated by a non-zero initial PDC value in that configuration context.

5.5.2 Contribution: Software Modifications For Single-Step Pipelining

The pipelining algorithm remains largely unchanged; the difference lies mainly in the preparation of the kernel data flow graph. The DFG edges representing operations which have internal state, have to be reserved for insertion into the first or last pipeline stage, as appropriate.

To prevent these from blocking the pipelining algorithm, all predecessors of the DFG edges reserved for the first pipeline stage must also be reserved for the first pipeline stage. So too must all of their predecessors, and so on (recursively).

Similarly, to avoid DFG edges not being assigned a pipeline stage, all successors of the DFG edges reserved for the last pipeline stage must also be reserved for the last pipeline stage. So too must all of their successors, and so on (recursively).

The pipeline stage assignment process begins by creating the first pipeline stage using the DFG edges that were reserved for the first pipeline stage. Then the pipelining algorithm is used to determine which DFG edges can be added to the current pipeline stage, and which require a new pipeline stage to be created in order to satisfy the timing constraint (and other constraints). Finally, the edges reserved for the last pipeline stage are added to the last pipeline stage that was created, unless doing so would violate the timing constraint, in which case a new pipeline stage is created for them.

As before, the timing constraint is adjusted if it is less than the critical path of any of the non-pipelineable data paths, which include the edges reserved for the first pipeline stage, and those reserved for the last pipeline stage.

5.5.2.1 Registers

Recall that registers appearing in the original kernel data flow graph, that are both read from and written to in the kernel, are placed in the first pipeline stage if possible. This is to ensure that their initial value (upon entering the kernel) is not corrupted during pipeline stage filling. However, if their final value (upon exiting the kernel) is important, placing the operation in the first pipeline stage results in the final value being corrupted during pipeline stage flushing. Also recall that the operations involved in updating the register's value constitute feedback, and must all be in the same pipeline stage, otherwise it would take more than one iteration for the new value to propagate through, leading to corruption. Therefore, the situation cannot be resolved by placing the input side of the register in the last pipeline stage.

The special-case solution to this problem for registers (i.e. preserving their final value upon exiting the kernel), is to duplicate the register so that the new value is written to the original register and another register at once. A chain of pipeline stage registers is then used to bring that duplicate value into the last pipeline stage, from where it will survive pipeline stage flushing. A new step is created after the kernel, to copy this final value back to the original register. By convention, this step is labelled the same as the kernel, but with the '`_finalise`' prefix appended. This is visualised in figure 5.9 for a typical kernel register that is being used as a counter.

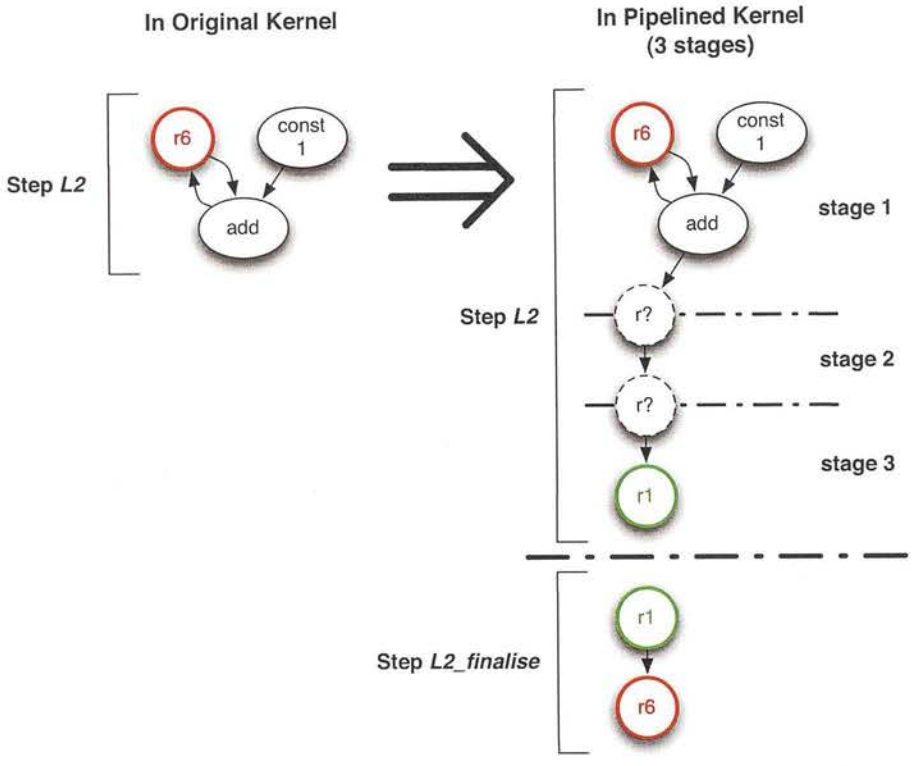


Figure 5.9: Construct used to preserve the final values of kernel registers upon exit from the kernel. This is used only where necessary. The kernel register ($r6$) is highlighted red. $r1$ (highlighted in green) is the register used to bring the final value out of the kernel. The required pipeline stage registers are shown with a dotted outline.

For k kernel registers, up to $k \times n$ additional registers would be required (where n is the number of pipeline stages). To minimise this overhead, the live register identification algorithm described in section 4.7.1 (on page 77) is used to determine which registers are actually live on entry to the next block after the kernel, so that only these need to have a delay chain created for them. In many cases, the final values of all the kernel registers are found to be irrelevant, so no delay chains and no *finalise* step are created.

As discussed at the beginning of this section, some kernel registers are written to from operations further down the critical path of the original kernel. If these are placed in the first pipeline stage, then the maximum possible pipeline depth is severely limited. The live register information is again used to identify which of these register's initial value is dead on entry to the kernel, which therefore allows them to be placed in any pipeline stage.

However, in a very small number of cases experienced, the initial value of some of these kernel registers is live. This prevents an adequate pipeline from being formed. A solution was devised to avoid this: additional logic is added to the chain of operations that supply the input of the kernel register. This additional logic consists of a multiplexer cell to choose between the newly calculated value, and the current (initial) value of the kernel register. The select signal of this

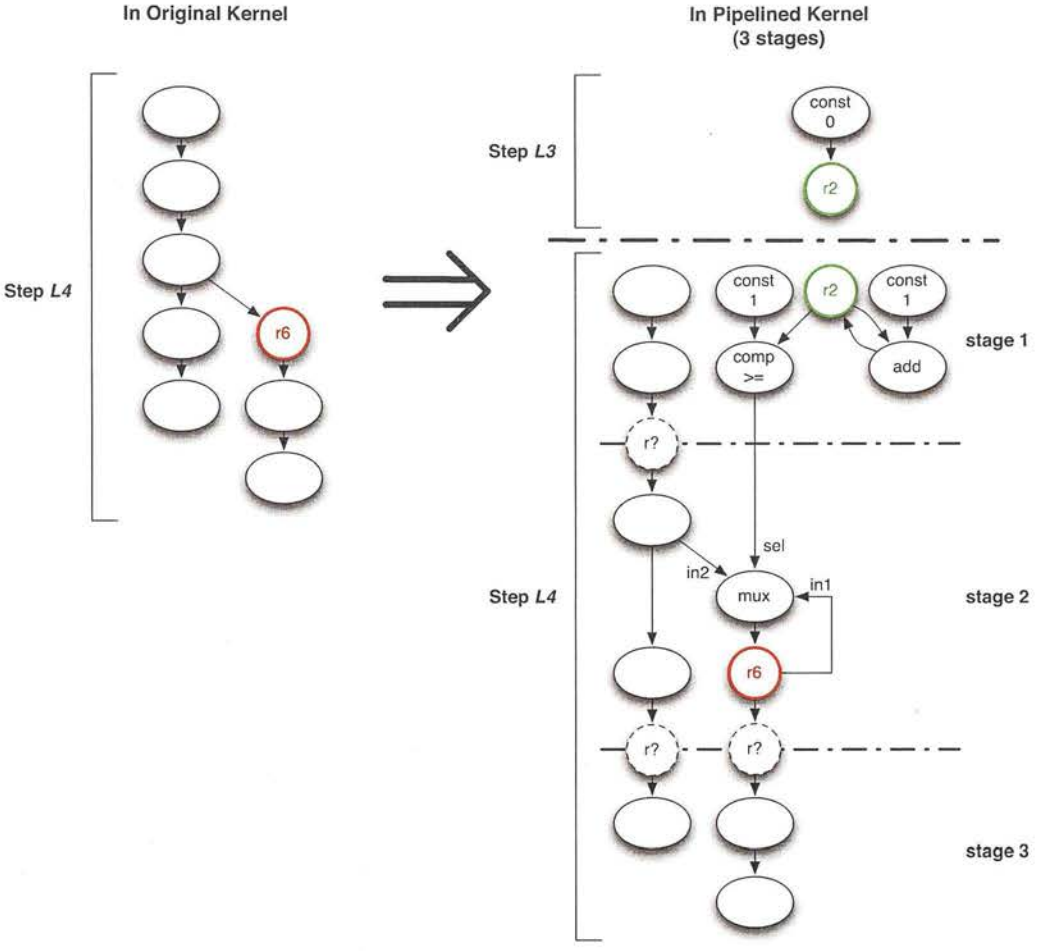


Figure 5.10: Construct for supplying the initial value to kernel registers on entry to the kernel, allowing them to appear in any pipeline stage. This is only required in rare cases. The kernel register is highlighted red. Pipeline stage registers are shown with a dotted outline. Register $r2$ (highlighted in green) is used as a counter to drive the mux select signal, and must be reset to zero in all steps that pass control to the kernel (`L3` in this example). This allows the kernel register to maintain its initial value until the first new value is ready.

multiplexer is then driven by a counter constructed out of another register and an adder cell, along with a comparator. The whole construct is shown in figure 5.10. The counter is used to make the mux make the kernel register hold its initial value during the pipeline filling iterations prior to the stage where the kernel register appears, and then switch to the newly calculated value thereafter. The counter is placed in the first pipeline stage, but directly drives the mux, despite the mux being in a later pipeline stage. Special logic in the pipelining algorithm is used to avoid inserting pipeline stage registers in this case.

5.6 Contribution: Automating The Choice of Timing Constraint

The previous sections in this chapter proposed a technique where pipelining would be performed based on a critical path constraint provided by the application developer. This section elaborates on this technique, by proposing how to automate the choice of critical path constraint, in order to maximise the real-life throughput whilst minimising resource usage.

The arbitrary operation chaining supported by the target architectures leads to a great variation in critical path length in different configuration contexts, as paths can be constructed involving long chains of a varying number of cells, and each type of cell has a different combinatorial delay. Ideally, each iteration of the configuration context should be allowed to persist for the time required for the results to stabilise on the operation(s) that lie at the end of the critical path. In order to avoid the overhead of asynchronous logic, a master clock is normally used instead, and the iteration ends on the next master clock cycle after the last results have stabilised, as can be seen in figure 5.11. To minimise the resulting idle time between these two events, it is desirable to minimise the period of the master clock. However, high clock frequencies come at the cost of power consumption and area. Therefore, a suitable compromise has to be made.

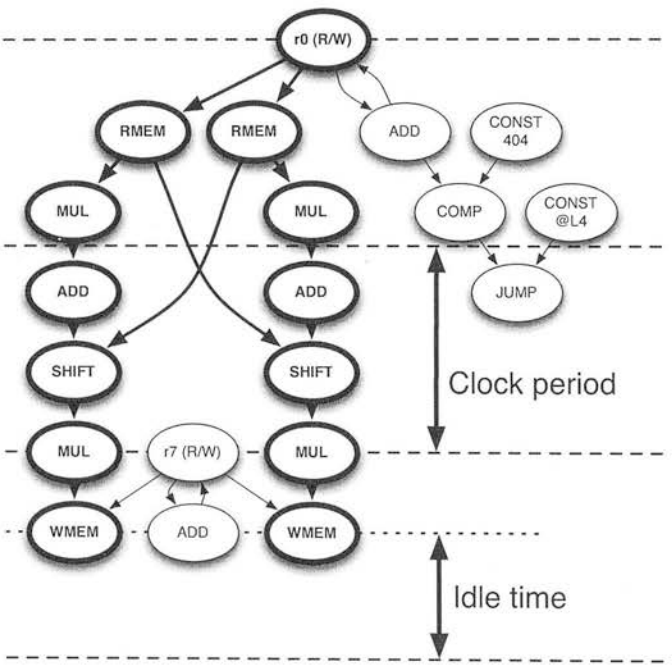


Figure 5.11: Idle time resulting from the master clock. The shorter the critical path of the kernel, the more effect this has. This particularly affects pipelined kernels.

Since pipelining reduces the critical path length of each iteration of the kernel loop configuration context, the quantisation introduced by the master clock frequency affects pipelined contexts more. Therefore, it is important to minimise the wasted time between the critical path stabilising and the next master clock cycle. This fact is used to aid the automatic choice of the timing constraint.

The timing constraint is initially chosen to be the minimum possible critical path length that a pipeline stage can consist of. This is determined by the length of certain data paths that cannot be split across pipeline stages. These include the jump condition logic determining when to finish the loop, and feedback loops that update a register or memory location[7]. The one with the longest critical path length is selected, and the value rounded up to the next integer multiple of the master clock period.

Then, pipeline stage allocation is performed using this critical path constraint. If a valid pipeline could be constructed, register allocation is performed. If there are sufficient registers available, then this pipeline geometry is used, since it will result in the highest possible iteration rate. Otherwise, the timing constraint is incremented by one master clock period, and the process continues. A natural end point exists where this value reaches the critical path of the non-pipelined kernel. If reached, the context is left non-pipelined.

A linear search of possible target critical path constraints could be quite slow, so an optimisation is to perform a binary search instead. The initial search space is that bounded by the minimum possible pipelined critical path and the original (non-pipelined) critical path. A binary search consists of performing multiple passes of splitting the current search space in half[8], choosing the upper half as the search space in the next pass if pipelining succeeds with that target, or the lower half otherwise. The search stops once the search space falls below the granularity of the master clock. A record is kept of the shortest target critical path constraint that resulted in a valid pipeline, so that this can be restored once the search ends.

For completely automatic pipelining, static analysis is used to identify configuration contexts that loop back to themselves (*kernels*). These are potential candidates for pipelining. The minimum consecutive iterations for a kernel defines the maximum depth to which it can be pipelined: the pipeline depth must not be less than the minimum execution count. This is used as a test during each iteration of the timing constraint selection algorithm, where a potential pipeline is checked for its depth not exceeding the minimum iteration count. If it does, then the geometry is considered invalid, and the algorithm continues with a larger timing constraint. Furthermore, for the multi-step pipelining technique (described in section 5.4), to take into account the cost of loading the new configurations from memory, the minimum iteration count value is artificially reduced by an arbitrary count, to weigh the algorithm in favour of only pipelining loops with significant iteration counts.

Static analysis or feedback-directed optimisation is used to determine the minimum possible iteration count for each kernel. Feedback-directed optimisation is used only if static analysis identifies that the iteration count is variable (and thus not statically analysable). For feedback-directed optimisation, the program is first executed in the emulator (chapter 3) prior to pipelining, and profiling information is fed back into the compiler. The number of consecutive iterations of each candidate is determined through the profiling results.

[7] where that register or memory location is both read from and written to in the same kernel.

[8] using that value as the next target critical path constraint.

5.7 Contribution: Support for Internally Pipelined Cells

The ability to pipeline complex kernels using the algorithm and techniques presented earlier in this chapter, means that in many cases, the critical path of the pipelined kernel is that of reading from a (pipeline) register, passing combinatorially through one or two functional cells, then writing to another (pipeline) register. This reduction in critical path is what leads to substantial increase in throughput, but also means that the combinatorial delay of each cell accounts for a much larger relative contribution to the critical path, and thus also the throughput. The difference between the combinatorial delay of each type of cell therefore becomes increasingly significant. For cells with large combinatorial delays such as dividers, multipliers, and random access to memory, these often cause a bottleneck in the step.

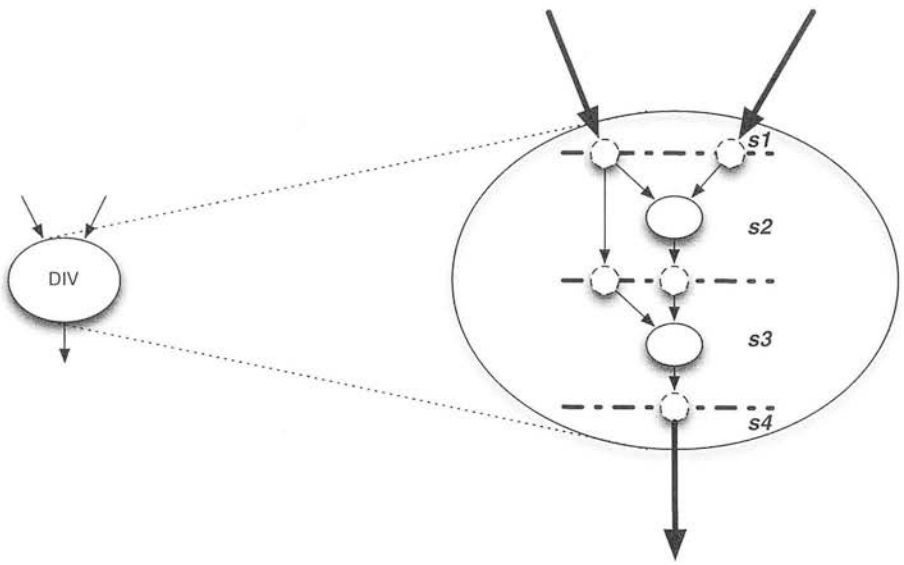


Figure 5.12: A divider cell internally pipelined to 4 stages. For minimum critical path contribution, reading from the inputs of the cell constitutes one pipeline stage (stage 1), and writing to the outputs of the cell constitutes one pipeline stage (stage 4).

The solution to this is to break up the internal logic in these cells into pipeline stages, as illustrated in figure 5.12. Unlike the dynamic pipelining talked about earlier in the chapter, this internal pipeline is fixed in hardware at design-time for the array, and cannot be altered.

Once internally pipelined, a cell can no longer be combinatorial—the result for the current iteration is delayed by several iterations[9]. So configuration contexts involving these cells have to be modified to take account of this. Both scheduling and pipelining are modified accordingly.

Since each cell type can support several configurations which correspond to different operations[10], and these operations can be of different complexity, each operation should be allowed to be pipelined to a different depth.

[9] the number of pipeline stages minus one.
[10] each represented by a different instruction.

5.7.1 Contribution: Scheduling Internally Pipelined Cells

When passing values into an internally pipelined instruction, the corresponding result does not appear at the output of the cell until several iterations later. The scheduling algorithm and data flow graph (DFG) data model that it operates on, as introduced in section 4.5 on page 67, assume that all operations—except registers—are combinatorial.

There are two ways to ensure that the operations that depend on the result of the internally pipelined instruction (with m pipeline stages) appear in the correct iteration:

- Split the basic block into multiple steps, making sure that the successors of the internally pipelined instruction appear m steps later than where the inputs were written to the cell.
- Create a single step from the basic block and pipeline that step, making sure that the successors of the internally pipelined instruction appear m stages later than where the inputs were written to the cell.

The data flow graph (DFG) data model is altered as follows: Instructions are now split into *slots*—each slot corresponding to an internal pipeline stage. Where previously each available instruction cell resource in the core was subdivided into available instruction slots[11], now each instruction slot is further subdivided into pipeline stage slots. Any or all of the pipeline stage slots for an instruction slot can be occupied in each step.

An example of this is shown in figure 5.13. The figure shows a hypothetical cell for accessing the contents of a stream memory (line buffer). The cell can be set up to read successive entries, one per iteration, automatically incrementing the address each time. To hide the memory latency, the memory access is internally pipelined (fixed in hardware). Since the address can be predicted beforehand and is not read from the data path in the same step, there is no need to expose the internal pipeline to the reconfigurable data paths—the cell can still be combinatorial (it has only an output). In this mode, the `STREAM_SET_ADDRESS` instruction sets the beginning address to start reading from, and begins fetching the first few values before the step ends. The next step in the program (usually a kernel) would then use the `STREAM_READ` instruction to fetch subsequent values from the stream, one per iteration. To be able to perform random access reads on the contents of the stream, one per iteration, the internal pipeline has to be exposed to the core. This is because the address cannot be predicted beforehand—it has to be read from the reconfigurable data paths. Therefore, the cell has to have both an input (the address) and an output (the result from an earlier iteration) active in the same step. The operations of setting the start address, sequential reading, and random reading are mutually exclusive—they cannot occur in the same step. However, the 3 stage internal pipeline of the random read instruction allows for up to 3 random read operations to be in-flight at once (if it appears in a pipelined kernel, as shown in figure 5.13(f)).

So, where previously an instruction would have a single DFG edge associated to it (like that shown in figure 5.14(a)), an internally pipeline instruction is represented by multiple DFG edges, one for each internal pipeline slot of that instruction (as shown in figure 5.14(b)). Only the first of these DFG edges has inputs, which correspond to the inputs of the instruction. Only

[11] each corresponding to a configuration representing a different type of instruction supported by that cell, where only one could be occupied in each step.

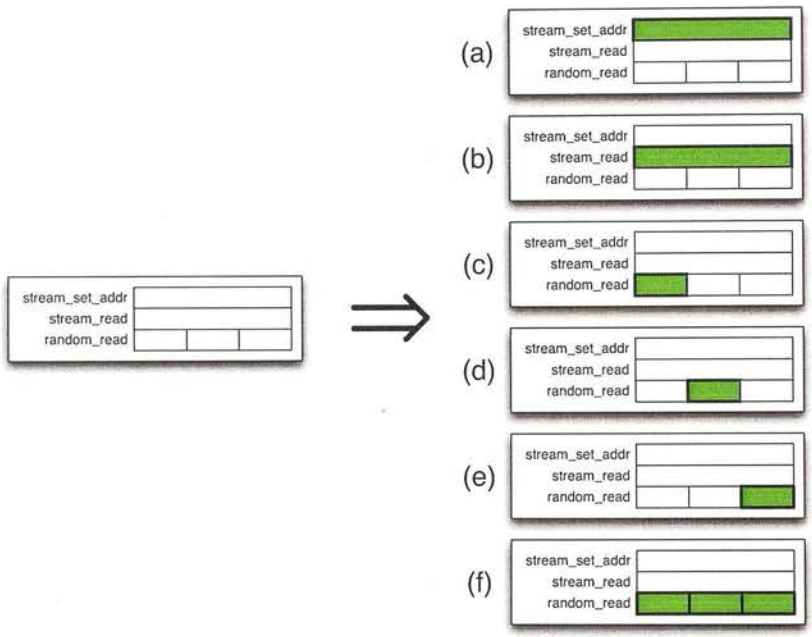


Figure 5.13: Instruction slots corresponding to an instance of a cell that supports 3 different types of instruction, one of which is internally pipelined. Valid examples of which slots can be filled (highlighted in green) in a configuration context are shown: (a) setting the start address for reading from a stream, (b) reading from a stream, auto-incrementing the start address, (c,d,e) random access read where the basic block has been split into multiple steps, (f) random access read in a pipelined kernel.

the last of these DFG edges has an output, which corresponds to the output of the instruction. All the other DFG edges for the instruction have no inputs or outputs. This prevents invalid connections from being made which tap into inaccessible internal partial results. However, in order to expose the dependencies between these edges,[12], *some step later* constraints are defined between them, as shown by the bold arrows in the figure.

All that needs modifying in the scheduling algorithm (section 4.9 on page 87) is the check for which cell resources are still available. The instances of the corresponding cell type are checked for unoccupied slots matching the edge that is being scheduled. In order for a cell instance to be considered available, it must now satisfy the following:

- It must match any explicit instance qualification given for the instruction.
- It must have no occupied instruction slots for other instruction types.
- The corresponding internal pipeline slot for the instruction type and internal pipeline stage must be available.

[12] to prevent them from being scheduled out-of-order.

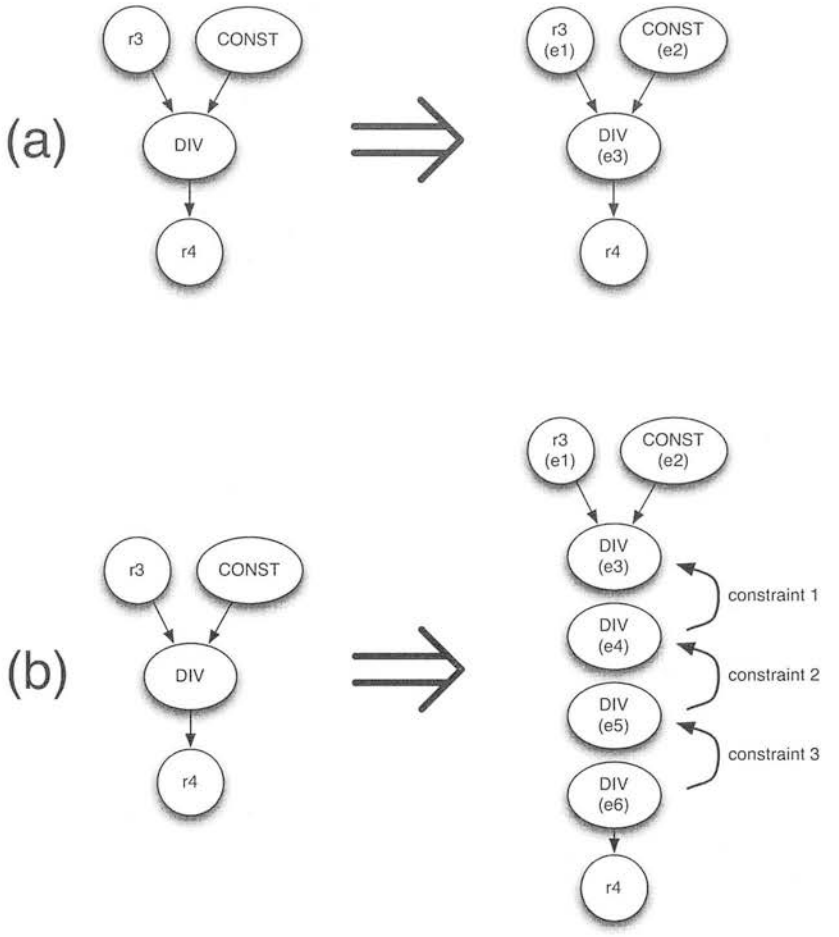


Figure 5.14: Assembly data flow graph (DFG) (left) and corresponding DFG edges (right) representing a data path containing: (a) a combinatorial divider cell, and (b) a 4-stage internally pipelined divider cell.

To prevent DFG edges representing the internal pipeline stages for a given instruction from being distributed across more than one cell instance, the instructions have to be modified prior to scheduling, binding them to an explicit cell instance. Because this involves performing cell instance allocation prior to scheduling, there is a danger that it could lead to premature starvation of a particular cell instance, creating more steps than strictly necessary. To reduce the chance of this, during DFG generation, a *potential bindings* map is generated for the instruction cell resources. This map records the number of instructions in the basic block that could potentially bind to each cell instance. Explicit cell instance qualifications are given to each internally pipelined instruction in the assembly in ascending order of potential binding count.

If the basic block is to be pipelined, then the *some step later* constraints are replaced with *same step or later* constraints, to avoid splitting the block into multiple steps. If pipelining subsequently fails, then the basic block must be re-scheduled, with the *some step later* constraints restored.

5.7.2 Contribution: Pipelining Kernels With Internally Pipelined Cells

The only modification to the pipelining algorithm necessary to support internally pipelined cells, is the addition of a *some stage earlier* constraint. The scheduling algorithm currently provides two types of constraints: *same step or later* and *some step later*. Normally, since pipelining is performed only on basic blocks that form a single step, the *some step later* constraint would never be encountered by the pipelining algorithm. The remaining constraints simply define a data flow dependency, similar to that defined by the connectivity through the data paths. The *same step or later* scheduling constraint is converted into a *same stage or earlier* pipelining constraint, with the operands reversed.

The new *some stage earlier* constraint is applied between the DFG edges representing the internal pipeline stages of an internally pipelined instruction. The pipelining algorithm takes into account this constraint when assigning pipeline stages, by incrementing the stage index where the edge could first be assigned to. Because instruction cell assignment has already been performed, and because the only dependencies between the DFG edges representing the internal pipeline stages are the aforementioned constraints, it is not possible for any other DFG edge to come between these DFG edges, inserting extra pipeline stages in between them. As a result, these DFG edges are correctly placed one stage apart, and the resulting data paths are seen to have been delayed by the correct amount, so the output feeds directly into the data paths of the appropriate pipeline stage. This approach also allows the pipelining algorithm to assign additional pipeline stage registers to the output of the internally pipelined cell, if required to feed additional pipeline stages in the dynamic (software-defined) pipeline.

During resource configuration, all the DFG edges representing the internal pipeline stages of a given instruction map to the same instruction cell,[13] resulting in the correct pipelined behaviour, with no pipeline bubbles.

As mentioned earlier, if a step containing internally pipelined operations cannot be pipelined, then the basic block must be re-scheduled, this time allowing step boundaries to be inserted between the place-holders for the internal pipeline stages for each internally pipelined instruction. Without this, insufficient iterations would be performed for the data to propagate through the internal pipelines, resulting in data corruption.

[13] filling all of its slots for that instruction.

5.8 Results

This section shows the results from experiments that were devised to demonstrate the following:

Section 5.8.1: Improvement in throughput v.s. cost in terms of additional registers and configuration contexts for two examples with very different minimum consecutive iteration counts, resulting from pipelining with the multi-step (section 5.4) and single-step (section 5.5) pipelining algorithms.

Section 5.8.2: Comparison of achievable throughputs when internally pipelined memory access cells are used (section 5.7) v.s. non-pipelined memory access cells.

Section 5.8.3: Behaviour of the automatic target critical path constraint identification algorithm, in terms of ability to achieve the highest real-life throughput, whilst minimising the register count required for that throughput.

5.8.1 Results: Dynamic Pipelining

These examples were implemented in C, targeting a 65nm RICA core with sufficient resources to implement the resulting kernels (i.e. around 250 cells and an abundance of registers). The master clock (RRC) period is 1.0ns. On-chip SRAM memory latency is 2.0ns, and the step load time is 20.0ns (no compression), with the capability to pre-fetch one step in advance. Typical kernels mapped to this core would have a critical path in the range of 20–80ns. Simpler steps may have critical paths in the range 7–20ns. The non-pipelineable data paths such as the program counter update chain (jump chain) are 4–7ns in length.

The first example (demosaic) was chosen as a typical data path intensive application, so as to be representative of the types of streaming applications that the target architecture was designed for. The module consists of one large data path with interdependences, to test the ability of the pipelining algorithm to insert stages without exceeding the target critical path constraint. This data path has a critical path several times that of the non-pipelineable data paths (e.g. the jump chain), which leaves room for an improvement in throughput. This situation is very common. As well as demonstrating the sort of throughput increase to be expected from pipelining, the example is to give a feel for the cost in terms of program memory size and register count for the two pipelining methods (multi-step and single-step pipelining).

The second example (DCT) was chosen as a special case to demonstrate some limitations of pipelining, when dealing with loops with low iteration counts. In particular, it is intended to show how the reduction in iteration time doesn't necessarily translate to a similar reduction in total execution time (which determines throughput), due to more iterations needing to be performed for filling and flushing. Furthermore, it was also a test to determine whether it is advisable to pipeline loops with low iteration counts.

5.8.1.1 Simple Demosaic

The first example is a 3-line demosaic filter [97], which involves interpolating missing colour components from the output from a colour filter array sensor. This is a high-throughput task normally done on-chip (integrated into the sensor) as part of a custom image signal processing pipeline. The filter was re-implemented on a reconfigurable instruction cell array, using the C language. Software optimisation techniques were used to reduce the filter kernel into a single basic block, small enough to fit onto the target architecture in a single configuration context. The filter kernel data flow graph is shown in figure C.1 on page 209, and the summary of the operations involved are given in table 5.1. Kernels of imaging filters such as this process an entire line of the image each time they are called, so the minimum consecutive iteration count is very high.

Resource	Instance count
add/comp	47
constant	14
logic	6
multiply	2
multiplexor	55
shift	6
jump	1
source	1
sink	3
streambuffer	2
register*	82

Table 5.1: Demosaic filter kernel resource requirements, in terms of instruction cells on the target architecture. *This register count does not include pipeline stage registers (since this is before pipelining has been applied).

The throughput of the resulting filter is given in the 2nd column of table 5.2. The other columns show the effect of pipelining the kernel using the multi-step pipelining method described in this chapter, for several target critical path lengths (timing constraints). Table 5.3 shows the effect of pipelining the same kernel using the single-step pipelining method, for the same timing constraints. The pipelined kernel can be seen in figure C.2 on page 210.

The results are based on the filter operating on an image size of 644×477 pixels. The kernel operates without interruption on an entire line, so in the non-pipelined case, it persists for 644 iterations. Pipelining incurs an increase in the number of iterations to be performed—i.e. $n - 1$ filling iterations and $n - 1$ flushing iterations, for an n stage pipeline. Multi-step pipelining performs these extra iterations using new configuration contexts that are executed in sequence; whereas single-step pipelining executes the kernel context for these extra iterations. The throughput is calculated based on the total execution time of the kernel per line, including these extra iterations of the kernel step itself, or the loading and executing of each epilogue or prologue step. This gives the real-life throughput performance, taking overheads into account. The throughput is graphed against target critical path constraint in figure 5.15.

Target critical path (ns)	None	35.0	20.0	10.0	8.0	6.0	5.0	4.0
Actual critical path (ns)	53.8	34.9	19.9	9.75	7.83	5.84	4.92	4.25
Line execution time (μs)	34.8	22.6	12.9	6.64	5.46	4.36	4.15	4.22
Pipeline stages	1	2	3	7	10	15	27	29
Additional registers	0	24	43	143	207	319	603	644
Additional contexts	0	2	4	12	18	28	52	56
Throughput (MPixels/s)	18.5	28.5	49.8	97.1	118	148	155	153
Speed-up	-	54%	169%	424%	538%	699%	739%	725%

Table 5.2: Throughput performance of the demosaic filter kernel before pipelining, and after *multi-step* pipelining—for a range of different target critical path length constraints. Additional register and program memory (contexts) resource requirements are shown. Inserting more pipeline stages results in a reduction in kernel critical path, leading to a maximum speed up of over 7×. However it also results in an increase in the number of steps. When the quantisation of the 1GHz master clock is taken into account, the last two columns result in the same iteration rate, but the latter has more steps, so the total execution time is longer, leading to a lower throughput.

Target critical path (ns)	None	35.0	20.0	10.0	8.0	6.0	5.0	4.0
Actual critical path (ns)	53.8	34.9	19.9	9.75	7.83	5.84	4.92	4.25
Line execution time (μs)	34.8	22.6	12.9	6.53	5.25	3.98	3.38	3.39
Pipeline stages	1	2	3	7	10	15	27	29
Additional registers	0	26	47	156	233	361	674	721
Additional contexts	0	1	1	1	1	1	1	1
Throughput (MPixels/s)	18.5	28.5	49.8	98.7	123	162	190	190
Speed-up	-	54%	169%	433%	562%	774%	929%	926%

Table 5.3: Throughput performance of the demosaic 3x3 filter kernel before pipelining, and after *single-step* pipelining. Compare with table 5.2. A single additional configuration context is produced in each pipelined case, to restore the final values of live output registers. A maximum speed up of nearly 10× is achieved here, compared to 7× in figure 5.2, despite the same number of pipeline stages and critical path. This is because the single-step pipelining avoids the overhead of loading additional steps for filling and flushing.

Both methods of structural-level pipelining can be seen to significantly increase the throughput, at the expense of extra registers (figure 5.16). The multi-step pipelining method incurs an additional overhead in terms of the program memory required for each new configuration context (prologue and epilogue). The last column in tables 5.2 and 5.3 shows that a natural throughput limit is reached, determined by the longest non-pipelineable entity in the data flow graph. In this case it is a multiplier being fed by pipeline registers, and writing directly to a pipeline register (see the zoomed-in portion of figure C.2). The critical path is the combinatorial delay of reading from a pipeline stage register (0.2ns), interconnect (1.44ns), internal delay of a multiplier (1.07ns), interconnect (1.44ns), and writing to a register (0.1ns)—giving a total of 4.25ns. This constitutes a speed-up of nearly an order of magnitude.

The resulting throughput can be seen to peak at some critical path slightly longer than the minimum possible (5ns v.s. 4.25ns). This is because the closest integer multiple of the 1ns master clock is 5ns, so both these geometries have the same iteration rate. However, because the latter has more pipeline stages, more iterations have to be performed per line.

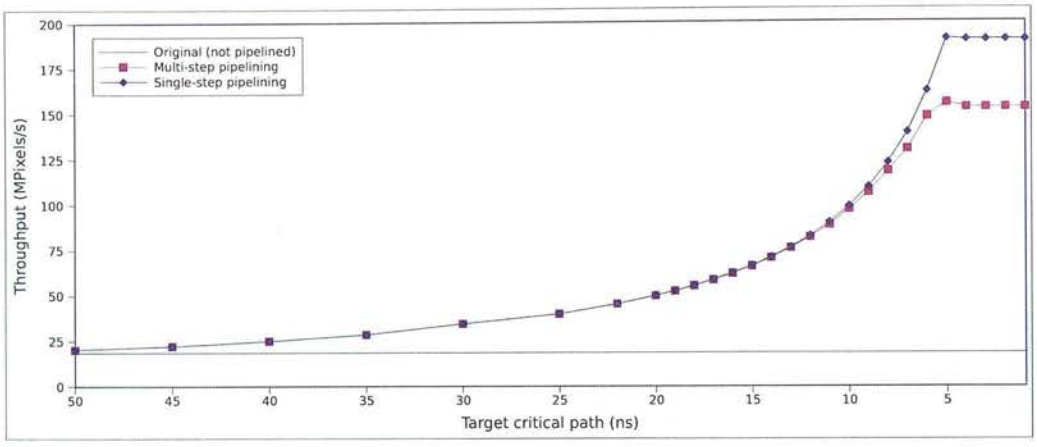


Figure 5.15: Measured throughput of the pipelined demosaic 3x3 kernel, for a range of target critical path length constraints. Both pipelining variants (multi-step and single-step) are shown. Throughput hits a wall once the non-pipelineable data paths dominate the critical path (in this case the jump chain, which is 4ns). Multi-step pipelining achieves a lower throughput for a given pipeline depth, due to the additional step loading times incurred at the beginning and end of each line.

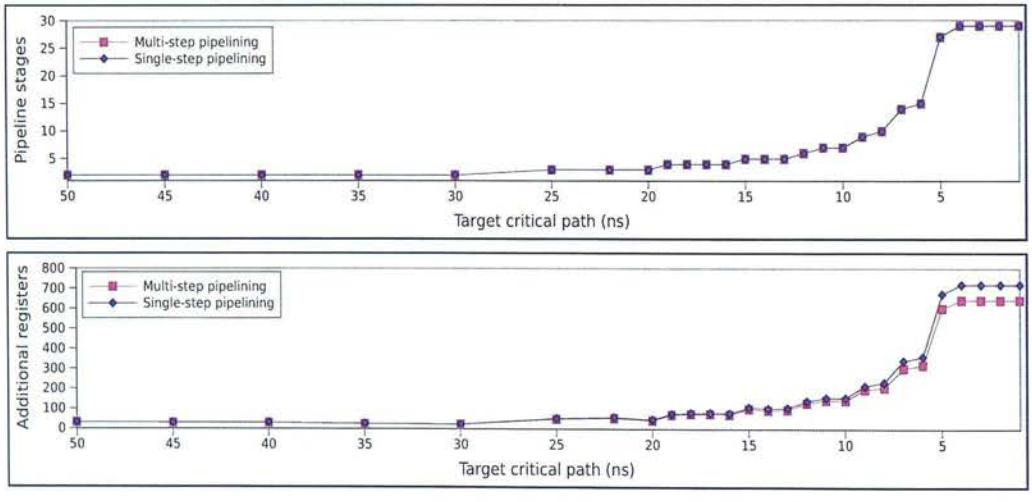


Figure 5.16: Pipeline stages and pipeline stage registers for each pipeline geometry resulting from a range of target critical path length constraints for the demosaic 3x3 kernel. Both pipelining variants (multi-step and single-step) are shown. Both variants construct the same pipeline geometry for each target critical path, but single-step pipelining requires more registers in order to feed initial values from the first pipeline stage and final values into the last pipeline stage (see section 5.5.2).

The maximum throughput achieved with multi-step pipelining is less than that achieved with single-step pipelining, as the increase in the number of pipeline stages causes the multi-step pipelining method to incur additional execution time overheads in terms of the time taken to load each additional prologue or epilogue step. As the number of pipeline stages increases

(shown in figure 5.16), the total step load times become an increasingly significant fraction of the total line execution time. Furthermore, as the iteration rate increases, the step pre-fetch mechanism becomes less able to hide the step loading time: pre-fetching of sequentially executed steps (like the prologue and epilogue) allows the next configuration to be loaded into shadow registers whilst the current context executes. Each of the prologue and epilogue steps are executed only once during each line; therefore pre-fetching is only able to hide the portion of the loading time that overlaps with the execution time (critical path) of the current step. The step loading time is a constant 20ns for the purposes of this example, so one would expect the throughput resulting from multi-step pipelining to start to lag behind that from single-step pipelining once the critical path drops below 20ns. The graph (figure 5.15) indeed shows this.

For very shallow pipelines, multi-step pipelining can have a slight advantage in throughput. This is because with shallow pipelines, the pipeline stages are often not very balanced, and as a result, the prologue and epilogue steps that contain only the first few pipeline stages can have a shorter critical path than the kernel itself. So long as these critical paths are greater than the step load time, this causes a reduction in total execution time compared with the single-step approach, where the kernel has the same critical path irrespective of which pipeline stages are currently active. The reduction in execution time achieved with multi-step pipelining is amortised over the execution time of the entire line, so the advantage is negligible unless the iteration count is quite low.

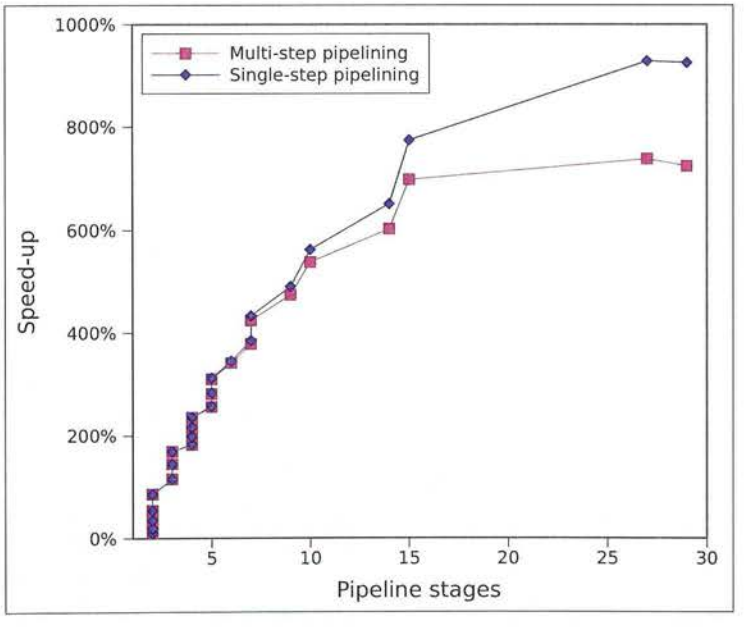


Figure 5.17: Improvement in throughput v.s. pipeline depth for the demosaic 3x3 kernel. The gains slowly decrease as more pipeline stages are added. This is the case for both pipelining variants (multi-step and single-step). This is caused by the additional lengths of interconnect needed to feed values to and from each set of pipeline stage registers, which extends the critical path.

Figure 5.17 shows that the speed-up is not linear with the number of pipeline stages—i.e. there are diminishing returns as the pipeline depth increases. This is mostly due to the idle time (see figure 5.11 on page 158) becoming an increasingly significant fraction of the per-iteration execution time, as the critical path decreases. Also, the internal delay of the pipeline stage registers[14] contributes to the total effective critical path, gradually requiring more pipeline stages to compensate.

The shapes of the pipeline stage and pipeline register graphs (figure 5.16) are very similar, which indicates that the register count is roughly proportional to the number of pipeline stages. Pipeline geometries resulting from single-step pipelining generally require more registers than from multi-step pipelining for the same target critical path, since more operations are constrained to be in the first or last pipeline stages, requiring more registers to bring those values to/from the pipeline stage where they are consumed/created. The more pipeline stages there are in total, the larger the gap in pipeline stages between the first pipeline stage and the value consumer, or the value creator and the last pipeline stage. Therefore the gap in register counts increases more than linearly between the two approaches.

With the dramatic increase in register counts evident in figure 5.16, the highest speed-ups can only be achieved with large cores.[15] However, as the size of the core increases, so too does the configuration size. The number of additional configuration contexts incurred by multi-step pipelining makes this very costly (easily an order of magnitude increase in program size). This is the situation where single-step pipelining comes into its own.

5.8.1.2 DCT

Reconfigurable architectures are capable of executing programs with complex control flow. This adds flexibility, allowing the same core to perform different tasks at different times. It also allows the use of algorithms too large to be mapped into a single context. This section shows an example of this common usage pattern, in a discreet cosine transform filter (8x8 DCT-II) [80] common in JPEG/MPEG image compression.

The entire 8x8 DCT is too large to implement in a single configuration context. Instead, this implementation of the filter performs an 8 element 1-D DCT for each row of input data, then performs another 8 element 1-D DCT for each column. The 1-D DCT is implemented as a kernel, which is shown in figure C.3 on page 211 (the operations are summarised in table 5.4). The kernel performs only 8 iterations for each of the two passes. This makes it a bad candidate for pipelining. Even so, the results in table 5.6 show that it is still possible to increase the overall performance of the filter using pipelining—a speed-up of 35% is demonstrated with single-step pipelining (the pipelined kernel can be seen in figure C.4 on page 211).

[14] and any additional interconnect leading to and from them.
[15] which have enough registers available.

Resource	Instance count
add/comp	43
constant	29
logic	0
multiply	16
multiplexor	0
shift	0
jump	1
read memory	8
write memory	8
register*	35

Table 5.4: DCT kernel resource requirements, in terms of instruction cells on the target architecture. *This register count does not include pipeline stage registers (since this is before pipelining has been applied).

Target (ns)	None	16.0	14.0	12.0	10.0	8.0	6.0
Actual critical path (ns)	16.8	15.0	12.5	10.5	8.51	7.99	6.71
Total execution time (ns)	483	487	441	415	437	483	467
Pipeline stages	1	2	2	2	3	4	4
Additional registers	0	5	13	9	18	31	25
Additional contexts	0	2	2	2	4	6	6
Throughput (MSamples/s)	133	131	145	154	147	133	137
Speed-up	-	-1%	10%	16%	11%	0%	3%

Table 5.5: Performance of a 2-D 8x8 DCT-II filter, for various *multi-step* pipeline geometries. The kernel only performs 8 iterations in each pass (rows or columns). Kernel critical path and total execution time are both shown, since in this case, a faster kernel doesn't necessarily lead to faster execution overall. This is because the step loading time for the additional fill and flush steps can become an appreciable fraction of the total execution time of the original kernel.

Target (ns)	None	16.0	14.0	12.0	10.0	8.0	6.0
Actual critical path (ns)	16.8	15.0	12.5	10.5	8.51	7.99	6.71
Total execution time (ns)	483	483	429	395	381	399	359
Pipeline stages	1	2	2	2	3	4	4
Additional registers	0	16	20	16	36	56	50
Additional contexts	0	0	0	0	0	0	0
Throughput (MSamples/s)	133	133	149	162	168	160	178
Speed-up	-	0%	13%	22%	27%	21%	35%

Table 5.6: Performance of a 2-D 8x8 DCT-II filter, for various *single-step* pipeline geometries. Compare with table 5.5. Single-step pipelining in this case imposes no additional configuration contexts, so no loading time overhead. This results in a consistent improvement in performance.

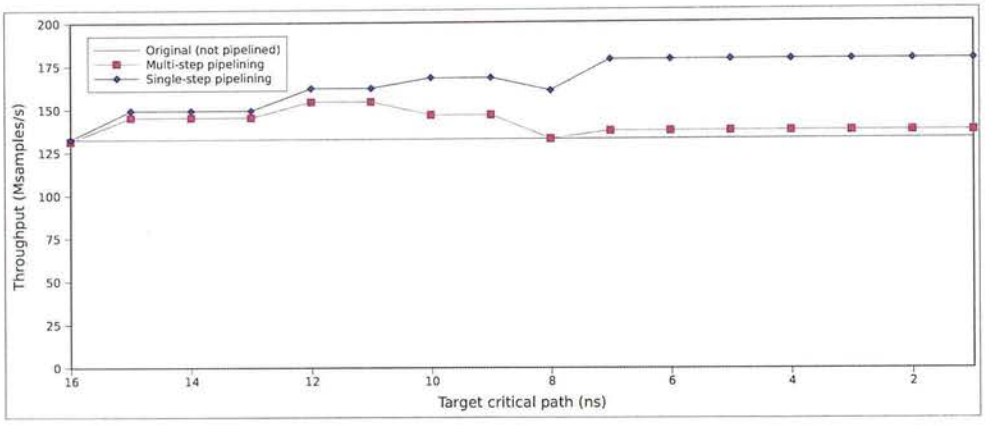


Figure 5.18: Measured throughput of the pipelined DCT kernel, for a range of target critical path length constraints. Both pipelining variants (multi-step and single-step) are shown. Only a minor improvement in throughput is seen, because the increase in iteration count imposed by pipelining is a substantial fraction of the total iteration count. Fluctuation is due to the relative effect of idle time (see section 5.6) and the increase in iteration count for every pipeline stage introduced. The additional step loading time incurred by multi-step pipelining can be seen here to nullify the effect of the reduction in critical path.

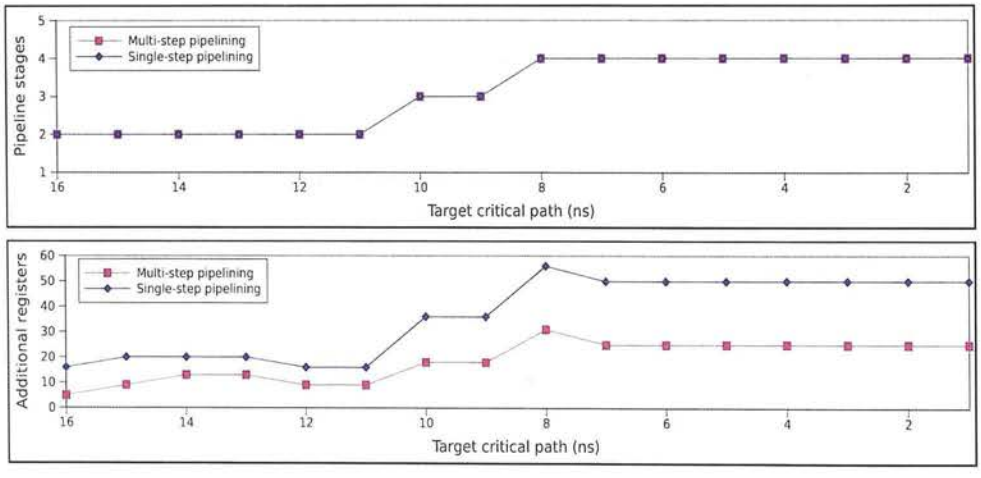


Figure 5.19: Pipeline stages and pipeline stage registers for each pipeline geometry resulting from a range of target critical path length constraints for the DCT kernel. Both pipelining variants (multi-step and single-step) are shown. The non-pipelineable data paths in this example have a critical path of 6.5ns. Additional registers are needed for single-step pipelining, despite the same pipeline geometry, in order to feed initial values from the first pipeline stage and final values into the last pipeline stage (see section 5.5.2). Register count isn't directly related to the number of pipeline stages; the shape of the data path determines this, as it depends on how many data paths are split at the particular point along the critical path where each pipeline stage is added.

The kernel has a short natural critical path: 16.8ns. This is less than the step load time. However, the kernel in this example relies on the shared random access data memory interface [61](section 3.5), which incurs additional dynamic delays with each memory access. The kernel uses 16-bit samples, reading-in 8 per iteration, and writing 8 per iteration. The architecture used for this example has 8 independent single-port 8-bit memory banks, allowing 4 samples to be read or 4 samples to be written in parallel. The memory accesses are serialised at run-time by the memory arbiter hardware. The data must be correctly aligned in order for this data parallelism to be achievable. This number of banks is quite high, imposing an area overhead for the arbiter logic. However, without this parallelism, this example becomes memory bandwidth constrained, in which case pipelining the data path wouldn't improve the iteration rate of the kernel.

Assuming this degree of memory parallelism is possible on the hardware, the actual per-iteration execution time of the kernel is given by the following equation:

$$t_{exec} = t_{cp} + r_{total}/n_{banks} \times t_{latency} + w_{total}/n_{banks} \times t_{latency}$$

Where:

t_{exec}: total kernel step execution time (one iteration).

t_{cp}: critical path delay rounded up to the next RRC cycle.

$t_{latency}$: data memory latency.

r_{total}: number of 8-bit read operations in the kernel.

w_{total}: number of 8-bit write operations in the kernel.

n_{banks}: number of independent 8-bit memory banks.

This means that the original (non-pipelined) kernel has a per-iteration execution time of:

$$t_{exec} = 17ns + 16/8 \times 2ns + 16/8 \times 2ns = 25ns$$

This is slightly larger than the step load time. In this example, the low iteration count means that the execution time of additional filling and flushing iterations can be significant compared to the total execution time of the kernel.

For multi-step pipelining, this leads to the best throughput being achieved when the overhead of loading and executing the additional steps is less than the reduction in per-iteration execution time of the kernel achieved through pipelining. Table 5.5 (and figure 5.18) shows this occurring with 2 pipeline stages and a data path critical path of 10.5ns, giving a 16% improvement in throughput. Taking into account the dynamic delays, the actual kernel per-iteration execution time is:

$$t_{exec} = 11ns + 16/8 \times 2ns + 16/8 \times 2ns = 19ns$$

Pipelining deeper than this decreases the per-iteration execution time to below the step loading time, which offsets the advantage.[16]

For single-step pipelining, this leads to the best throughput being achieved when the overhead of the additional iterations of the kernel is less than the reduction in per-iteration execution time of the kernel achieved through pipelining. Table 5.6 (and figure 5.18) shows this occurring with 4 pipeline stages and a data path critical path of 6.71ns, giving a 35% improvement in throughput. This is the deepest pipeline possible, with the critical path being determined by a non-pipelineable feedback loop involving a counter.

When single-step pipelining, reading from memory must occur in the first pipeline stage, and writing to memory must occur in the last pipeline stage. This means that more registers are needed for a given pipeline depth.[17] It also means that each iteration of the kernel during pipeline filling will have only memory reads, and each iteration of the kernel during pipeline flushing will have only memory writes. Thus the dynamic delay during filling or flushing is half that during normal kernel loop iterations. This slightly offsets the effect of the increase in the number of iterations, which is why the point of highest throughput occurs at a higher pipeline depth than with multi-step pipelining. It is also the reason for the higher maximum achievable throughput with single-step pipelining.

5.8.2 Results: Internally Pipelined Cells

As in the previous section, the example in this section was implemented in C, targeting a 65nm RICA core with sufficient resources to implement the resulting kernel (i.e. around 250 cells and an abundance of registers). The master clock (RRC) period is 1.0ns. On-chip SRAM memory latency is 2.0ns, and the step load time is 20.0ns (no compression), with the capability to pre-fetch one step in advance. A typical kernel will have a critical path in the range 20–80ns. Non-pipelineable data paths such as the jump chain are around 4–7ns.

The intention of this experiment was to demonstrate the effectiveness of providing internally pipelined cells. With internally pipelined cells, operations that would otherwise contribute to the non-pipelineable data paths that determine the minimum achievable pipelined critical path, can be amortised over multiple iterations, removing them from the critical path. The most common case of this is with memory access, so a memory bandwidth intensive streaming application was chosen.

[16] since the loading cost can only be amortised over the very small number of kernel iterations.

[17] to bring the values into the stages where they are used.

Resource	Instance count
add/comp	61
constant	26
logic	24
multiply	12
multiplexor	30
shift	60
jump	1
source	6
sink	6
stream buffer*	12
read memory*	12
register**	201

Table 5.7: Gamma correction filter kernel resource requirements, in terms of instruction cells on the target architecture. *Mutually exclusive, for the two different implementations. **This register count does not include pipeline stage registers (since this is before pipelining has been applied).

Target critical path (ns)	None	30.0	20.0	15.0	10.0	8.0	6.0	5.0
Actual critical path (ns)	44.8	27.9	19.3	14.9	9.82	7.81	5.82	5.37
Line execution time (μs)	30.7	19.9	14.1	10.9	8.40	7.14	5.88	5.29
Pipeline stages	1	2	3	4	6	9	13	20
Additional registers	0	30	43	65	106	155	237	338
Additional contexts	0	0	0	0	0	0	0	0
Throughput (MPixels/s)	41.6	64.4	90.5	117	152	179	218	242
Speed-up	-	55%	117%	181%	266%	330%	422%	481%

Table 5.8: Performance of the gamma correction filter kernel before pipelining, and after pipelining, using combinatorial memory operations (`RMEM`). Results are given only for the (unrealistic) hardware where all reads are performed in parallel. This gives a fairer comparison with table 5.9. The number of pipeline stages, critical path, and throughput increase in the usual manner as the target critical path constraint is tightened, achieving a maximum speed-up of nearly 5×.

Target critical path (ns)	None	30.0	20.0	15.0	10.0	8.0	6.0	5.0
Actual critical path (ns)	45.4	28.0	19.3	14.9	9.82	7.81	5.82	4.92
Line execution time (μs)	71.0	18.0	12.9	9.87	6.60	5.30	4.00	3.38
Pipeline stages	1/4*	5	6	18	19	22	26	33
Additional registers	0	84	108	329	342	406	471	590
Additional contexts	3*	0	0	0	0	0	0	0
Throughput (MPixels/s)	18.0	70.9	99.1	130	194	241	320	379
Speed-up	-	294%	450%	620%	977%	1239%	1673%	2004%

Table 5.9: Performance of the gamma correction filter kernel before pipelining, and after pipelining, using internally pipelined memory operations (`SRBUF_RAM`). Compare with table 5.8. *Without pipelining, the kernel has to be split into multiple steps to compensate. As a result, the non-pipelined throughput is very low, and thus distorts the speed-up values. Comparing the line execution times instead, this version achieves a pipelined throughput 35% higher than in table 5.8. This is due to the memory latency being hidden by overlapping it across multiple iterations, compared to the `RMEM` example where the latency has to appear in a single iteration, extending its critical path.

The example used in this section is a gamma correction module [81] from a typical image signal processing pipe. The complexity is shown in table 5.7. Gamma correction is applied in this case to two RGB pixel streams at once. Each sample is 16 bits. There are two 16-bit table look-ups required per sample—one look-up from each of the two 64 entry tables (base and gradient). This means that 12 16-bit memory operations are required per iteration of the kernel. The input and output streams are implemented via separate data interfaces, so do not consume data memory bandwidth. The table look-ups impose a significant demand on memory bandwidth. Two implementations are compared here: normal combinatorial data memory interface v.s. multiple independent stream/line buffers in random access mode (internally pipelined).

As mentioned in the previous section, the normal data memory interface hardware on RICA provides arbitration to serialise conflicting memory operations. This allows the memory bandwidth to be increased slightly, whilst still maintaining a single memory address space—i.e. it is a trade-off between ease of programming and bandwidth/area. Area grows exponentially as more memory banks are added to increase the bandwidth. The interface is combinatorial: the address is sampled in the current iteration, then at the next clock edge, that address is accessed in the memory interface, and the value stored there is returned for use in the current iteration. This effectively causes the critical path of the step to be increased by the memory latency times the number of operations that are queued, plus some idle time to round up to the next master clock period.

The first implementation of the kernel uses the normal data memory interface, represented by the `RMEM` instruction. The kernel is shown in figure C.5 on page 212, with resources summarised in table 5.7. The same code is run on two different variants of the hardware: a realistic interface consisting of 8 independent 8-bit banks, allowing 4 16-bit read operations in parallel; and an unrealistic interface consisting of 24 independent 8-bit banks, allowing all 12 16-bit read operations to occur in parallel. In the first hardware variant, each iteration of the kernel requires three sequential reads from each bank. In the second variant, each iteration requires only one read from each bank. This allows for a more fair comparison with the stream buffer implementation, described next.

A second implementation of the gamma correction kernel uses domain-specific stream buffers (line memories), with corresponding interface cells in the core, which are represented by the `SRBUF_RAM` instruction. A typical ISP core would contain many of these, so it is possible to achieve very high memory bandwidths for small data sets like the look-up tables in this example. These stream buffers are designed to be accessed sequentially, where each successive location is automatically de-referenced and returned at the beginning of each iteration of a kernel. This hides the memory access latency by fetching the next sample whilst the current iteration of the kernel is executing. This is only possible because the next address is known in advance. In order to perform random access whilst still being able to hide the memory latency, the operation has to be pipelined—i.e. the result appears at the output of the cell several iterations later. In the hardware used here, the `SRBUF_RAM` instruction is pipelined into 4 stages, meaning that the result appears 3 iterations after where the address was sampled.

Tables 5.8 and 5.9 and figure 5.20 show the relative performance of each of these scenarios, over a range of pipeline target critical path constraints (image size 640x474). Only single-step pipelining is demonstrated here. The pipelined kernel data flow graphs can be seen in figure C.6 on page 213 and figure C.8 on page 215.

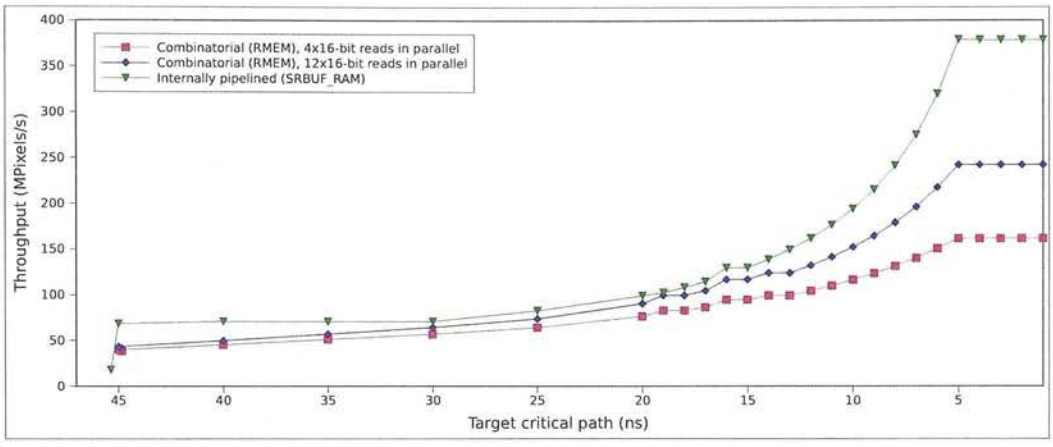


Figure 5.20: Measured throughput of the non-pipelined and single-step pipelined gamma correction kernel, for a range of target critical path length constraints. Compares combinatorial memory read operations (RMEM) (with two different memory interface characteristics) and internally pipelined read operations (SRBUF_RAM). Without pipelining, the internally pipelined cells lead to a spuriously low throughput, due to the kernel having to be split into multiple steps in order to bring the output into sync with the input. The throughput ceiling in the RMEM cases is determined by the memory latency (or a multiple of this when the reads have to be staged), as the memory accesses must occur entirely within one iteration (and pipeline stage). Memory latency is distributed across multiple iterations in the SRBUF_RAM case, so the throughput ceiling there is determined by other non-pipelineable data paths such as the jump chain. A knee can be seen at about 16ns, where integer multiples of the memory latency, master clock, and non-pipelineable data paths in the step happen to align.

The data points at the far left of the graph show the kernels when not pipelined. Note that the performance of the implementation using the internally pipelined stream buffer cells is spuriously low when the kernel is not pipelined. This is because the kernel must be split into multiple steps (4) in order to take account of the cycles of latency resulting from the internal pipeline in the interface cells (shown in figure C.7 on page C.7). This makes the kernel program memory bandwidth limited, since the core must be reconfigured four times per iteration of the kernel. It also increases the total critical path[18], due to the middle steps which just wait for data to propagate through the corresponding pipeline stage of the internally pipelined cells.

As can be seen in figure 5.21, when pipelining is enabled, the implementation using `SRBUF_RAM` pipelines to a minimum of 5 stages, in order to fit around the internal pipelining of the interface cells—this roughly translates to one stage containing the logic that generates the address, then 4 stages corresponding to the internal pipeline, with the last stage also containing logic that operates on the result.

[18] which is the sum of the critical paths of each step.

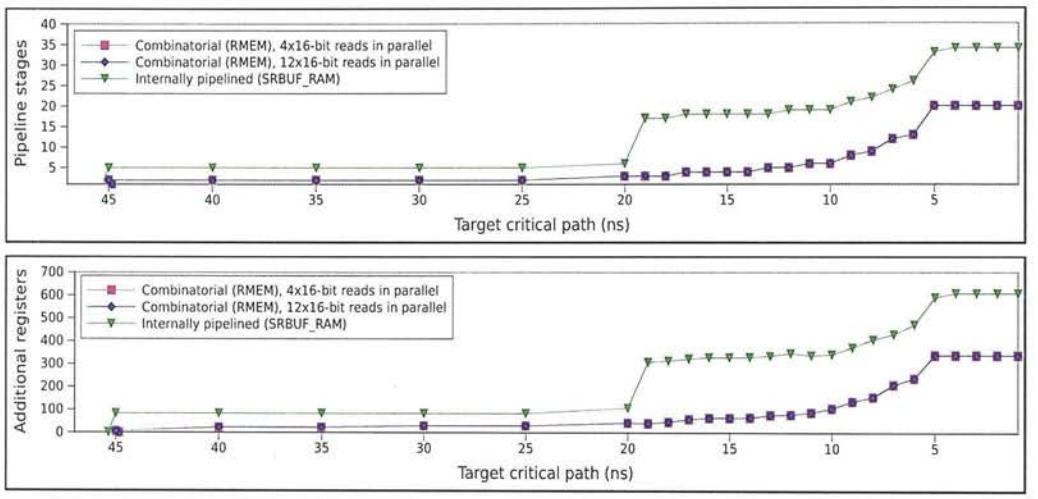


Figure 5.21: Pipeline stages and pipeline stage registers for each pipeline geometry resulting from a range of target critical path length constraints for the gamma correction kernel. Compares combinatorial read operations (RMEM) and internally pipelined read operations (SRBUF_RAM). The kernel using SRBUF_RAM when not-pipelined, has to be split into multiple steps. This is represented by a fractional pipeline stage count. With SRBUF_RAM, a minimum pipeline of 5 stages is needed to work around the internal pipeline of the memory access cell. A sudden jump in pipeline stages (and thus also pipeline stage registers) is seen going from 20ns to 19ns, where the pipeline stages introduced to work around the internal pipeline of the cells are insufficient to meet the critical path constraint, and so the data paths suddenly also need to be pipelined. This is followed by the usual exponential rise, until the upper limit is reached (caused by other non-pipelineable data paths).

For any given (achievable) target critical path, the actual performance of the implementation using combinatorial memory access cells (RMEM) is lower than that with the synchronous memory access cells (SRBUF). This is due simply to the memory latency extending the critical path. As a result, the version where reads have to be performed sequentially in batches, has lower performance than the one where they are all performed in parallel. The performance curves are a very similar shape, where the performance gap widens as the (constant) memory latency becomes an increasingly significant fraction of the total execution time of an iteration of the kernel.

Figure 5.22 shows how the addition of pipeline stages affects the speed-up (pipelined throughput compared to non-pipelined throughput). Both scenarios using combinatorial memory access cells show a logarithmic improvement. The relationship with internally pipelined cells shows a similar logarithm, but with a steep start and a step change shortly after. Below this step change, there is an artificially high pipeline stage count (high target critical paths), to work around the fixed internal pipeline depth of the memory access cells. This pipeline stage count jumps up once the kernel pipeline geometry begins to become more compatible with the internal pipeline geometry of the cells.

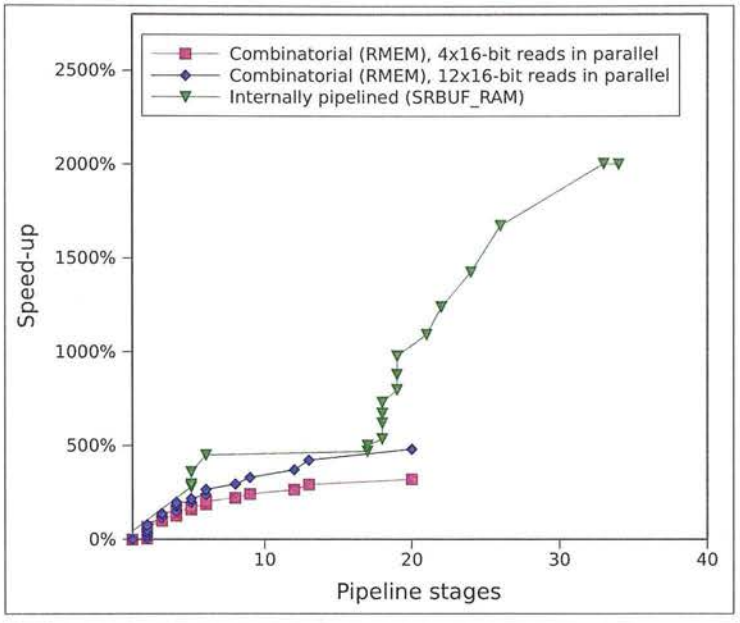


Figure 5.22: Improvement in throughput v.s. pipeline depth for the gamma correction kernel. The gains decrease logarithmically as more pipeline stages are added. This is due to the additional interconnect length introduced by inserting pipeline stage registers. The SRBUF_RAM case shows a step change corresponding to the knee seen in figure 5.21.

The minimum possible pipelined critical path is lower with internally pipelined memory access cells, since (as with other synchronous cells) the combinatorial delay is very short—the output of the cell behaves just like reading from a register; inputs to the cell are simply sampled at the end of the iteration, like when writing to a register. Therefore, the internally pipelined memory access cells show a significant performance advantage even over the unrealistic case where the combinatorial memory access interface has equal memory bandwidth. The cost of using internally pipelined cells comes in the form of register counts, since more pipeline stages are needed to keep the data paths involving combinatorial cells in sync with the data paths involving internally pipelined cells. However, in large cores suitable for image signal processing, registers are in abundance.

5.8.3 Results: Automatic Timing Constraint

This section demonstrates the effect of clock quantisation on the maximum achievable pipelined throughput, by pipelining examples on hardware with different master clock periods. The algorithm for the automatic choice of timing constraint (section 5.6) takes advantage of this effect to determine what the minimum achievable pipelined critical path should be, and uses that as the target. To make the effect of clock quantisation more pronounced, a slower technology process (180nm) was used, to make the resulting idle time a higher fraction of the pipelined critical path. Two applications were tested to demonstrate that the effect is not application dependent.

The single-step pipelining algorithm with automatic choice of timing constraint was applied to two real-life applications: a 7-line Hamilton demosaic filter [98], and a multiplication-based iterative software division algorithm. The demosaic involves interpolating missing colour components from the Bayer output of a colour filter array sensor. Division on a per-pixel level is used as part of many commercial noise reduction filters. Both are high-throughput tasks normally done on-chip as part of a custom image signal processing (ISP) pipeline, used in modern digital cameras and mobile phones. Both kernels were implemented on a reconfigurable instruction cell-based processor [5] (180nm timing figures), using the C language. Software optimisation techniques were used to reduce the main kernel in each case into a basic block small enough to fit onto the target architecture in a single configuration context. Both example kernels produce a single output pixel per iteration.

Master clock period (ns)	20.0	15.0	10.0	5.0	3.0	2.0	1.0
Pipeline stages	5	7	5	7	9	9	11
Pipeline stage registers	80	123	80	123	153	153	189
Min. possible constraint (ns)	10.95	10.95	10.95	10.95	10.95	10.95	10.95
Non-pipelined critical path (ns)	77.0	77.0	77.0	77.0	77.0	77.0	77.0
Pipelined critical path (ns)	19.8	14.65	19.8	14.65	11.55	11.55	11.00
Improvement in critical path	389%	526%	389%	526%	667%	667%	700%
Non-pipelined iteration time (ns)	80.0	90.0	80.0	80.0	78.0	78.0	77.0
Pipelined iteration time (ns)	20.0	15.0	20.0	15.0	12.0	12.0	11.0
Improvement in iteration time	400%	600%	400%	533%	650%	650%	636%
Pipelined throughput (MPixels/s)	50.0	66.6	50.0	66.6	83.3	83.3	90.9
Speed-up	400%	600%	400%	533%	650%	650%	636%

Table 5.10: Performance of the Hamilton demosaic filter kernel before and after automatic pipelining, for a range of different master clock periods. See section 5.8 for an explanation of the results.

The performance of the pipelining for both cases is shown in figure 5.23, and some additional details are given for the Hamilton demosaic in table 5.10. The independent parameter in these experiments is the *master clock (RRC) period*—NOT the pipeline timing constraint (which was used in the previous experiments). This represents a physical difference in the hardware, rather than just a compile-time setting in the tools. The experiments in the previous sections show that simply pipelining to a smaller target pipeline timing constraint doesn't necessarily lead to an improvement in iteration rate, but does lead to an increase in resources. For example in table 5.3, a 4.0ns target leads to a deeper pipeline (more stages) than a 5.0ns target, yet the iteration rate is the same. This is due to quantisation: the error between the time taken for each data path fragment in each pipeline stage to complete and the closest integer multiple of the master clock period. The automatic timing constraint algorithm chooses a target which results in the least quantisation. The experiments in this section show the effectiveness of this, with different levels of quantisation—i.e. different master clock periods.

The main trend to notice is the ability for the maximum achievable iteration rate (after pipelining) to generally increase as the master clock frequency is increased. Since the same underlying data path is used in each case, the non-pipelined critical path length is constant. The iteration time of the non-pipelined data paths is just the critical path length rounded up to the next integer multiple of the master clock period. As the master clock period is decreased, the algorithm

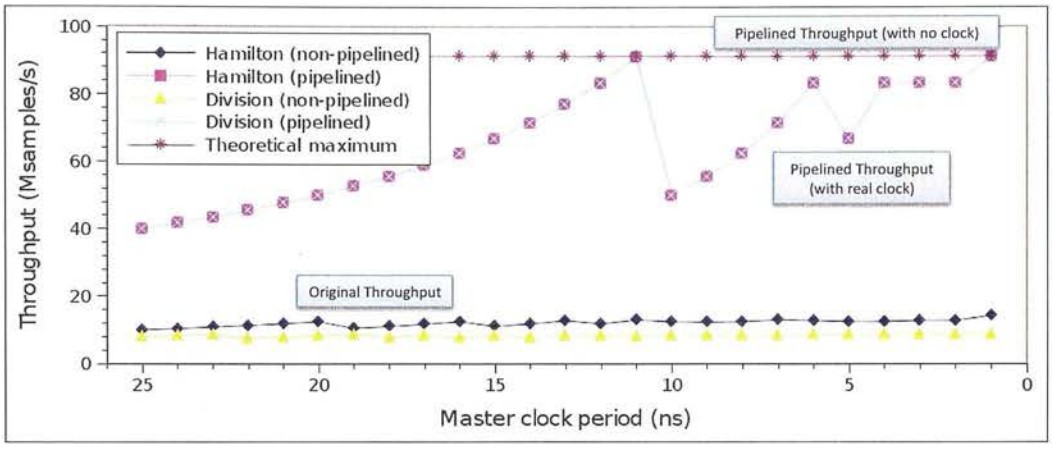


Figure 5.23: Throughput before and after automatic pipelining, for a range of different master clock periods, for two pixel-level code examples: Hamilton demosaic and iterative software division. The theoretical line shows what could be achieved if the master clock were of infinite frequency, based on the longest indivisible critical path (the iteration control logic in both of these cases). The throughput of the non-pipelined cases are determined by the critical path of that particular kernel, whereas the throughput of the pipelined cases are determined by the minimum integer multiple of the master clock able to cover the non-pipelineable data paths. The throughput of the non-pipelined cases vary only slightly, since the change in idle time caused by different master clock frequencies represents only a small fraction of the kernel critical path. The pipelined cases however show a pronounced exponential trend related to the idle time as a fraction of the master clock period, which wraps around whenever the pipelined critical path coincides with an integer multiple of the master clock period.

is able to produce a pipeline with a critical path closer to the theoretical minimum[19]. However, the number of pipeline stages required to do this increases in a faster than linear fashion. This is another effect of quantisation: as the pipeline stages get shorter, the relative size of the indivisible units being pipelined[20] increases compared to the resolution of the master clock. The algorithm does well in minimising this effect, and the percentage improvements with and without the effect of the master clock are relatively close in all cases.

The pipeline geometries contructed for each master clock frequency setting are shown in figure 5.24. Both examples show identical post-pipelining throughput (iteration rate), as both cases have the same longest indivisible critical path—corresponding to the iteration control (jump) logic (shown by the theoretical line in figure 5.23). There are no data dependencies or other constraints limiting the potential for pipelining in either example. If data dependencies, feedback loops, or other constraints were present, these would be reflected by a larger indivisible critical path. The shorter the indivisible critical path, the more important the behaviour of the automatic pipelining algorithm.

[19] as dictated by the indivisible data paths such as feedback loops, and the jump condition chain.

[20] i.e. the internal delays of each cell and section of interconnect.

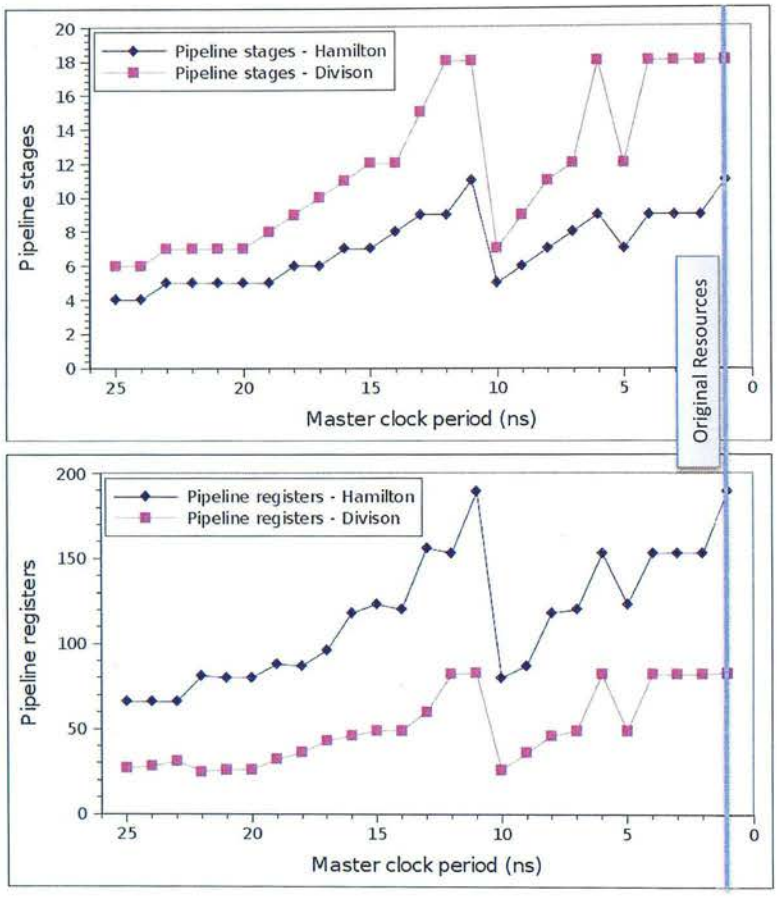


Figure 5.24: Pipeline geometry from automatic pipelining, for a range of different master clock periods, for two pixel-level code examples: Hamilton demosaic and iterative software division. The two examples have a different non-pipelined critical path, so the number of pipeline stages differ, despite the pipelined throughput being the same (see figure 5.23). The automatic timing constraint algorithm can be seen to reduce the number of pipeline stages accordingly, when the idle time prevents a higher throughput being achieved. This has a significant effect on reducing register consumption. As before, there is some randomness in the number of registers required for a given pipeline depth, as this depends on where along the critical path the registers have been inserted (i.e. how many data paths span across pipeline stages).

The resource-saving effect of the algorithm can be seen to come into effect each time the current integer multiple of the master clock frequency drops below the indivisible critical path length. This makes the iteration rate curve appear to wrap around each time it tries to cross the theoretical maximum iteration rate line. One such boundary is identified in figure 5.25. By extending the length of the pipeline stages up to the next master clock period, the number of registers is minimised, which avoids needless congestion on the interconnect. The reduction in the number of pipeline stages reduces the configuration size and the latency, since fewer filling and flushing iterations need to be performed.

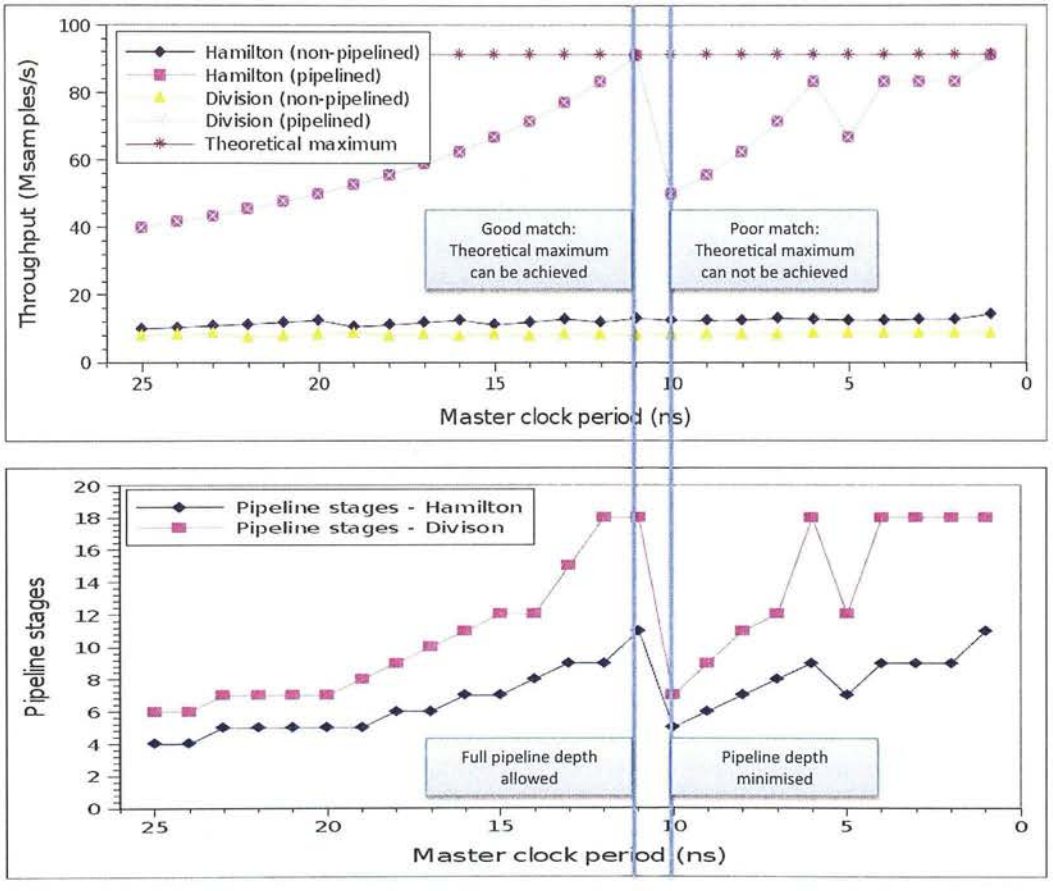


Figure 5.25: The graphs of figures 5.23 and 5.24 to show the correlation between them. When the clock frequency is such that maximum theoretical throughput can be obtained, the appropriate number of pipeline stages are created, as shown by the line on the left. The line on the right shows that when the clock frequency is such that the maximum achievable throughput is significantly less than the theoretical maximum, the algorithm relaxes the number of pipeline stages—and thus saves resources—without compromising the achievable throughput.

5.9 Summary

This chapter described algorithms used to drastically improve the performance of compute-intensive loops. The data paths of the configuration context corresponding to a loop body can be split into pipeline stages. This decreases the critical path of the loop, increasing the iteration rate, and thus the throughput, at the cost of additional registers. For large cores, pipelining can lead to near ASIC levels of performance.

The first method (*multi-step pipelining*) is a software-only solution that uses dynamic reconfiguration to perform pipeline filling and flushing (by adding configuration contexts with some pipeline stages omitted).

A second method (*single-step pipelining*) was proposed which removes the need for additional configuration contexts, at the cost of some flexibility and minor modifications to the hardware. This is particularly advantageous on large cores, where the number of pipeline stages and the memory required to store a configuration context are both large.

Multi-step pipelining works well with small cores, where register counts are low, and the cost of additional configuration contexts is low. Single-step pipelining is most suited to large cores, where registers are in abundance, and where very deep pipelines can achieve significant gains in throughput. Single-step pipelining is also of use in loops with low iteration counts.

An algorithm for completely automating the task of pipelining was demonstrated, which automatically chooses a suitable pipeline target critical path constraint for the pipeline stage assignment algorithm to operate on, such that the maximum possible throughput is achieved with minimal cost in registers.

Further improvements were demonstrated by the addition of support for internally pipelined cells. This allows high-latency operations such as memory access to be amortised over multiple iterations, reducing the critical path, and thus further increasing the throughput. Dynamic pipelines are constructed around the internal pipeline depth of each cell. The cost of this is generally a small increase in registers, as additional pipeline stages are often needed to bring other values into sync with the results of internally pipelined operations, when the data paths in a configuration context happen to not line up well with the internal pipeline depth of the cells. Internally pipelined cells are also inefficient when used without pipelining, as the internal pipeline depth has to be compensated for by the addition of configuration contexts instead of pipeline stages.

The next chapter concludes the thesis by restating what was done, what was shown, its significance, and contribution to knowledge.

Chapter 6
Conclusions

The overall objective of the work presented in this thesis was to develop tools to allow dynamically reconfigurable computing architectures to be programmed and worked with in a manner similar to how microprocessors are. The tools must map high-level ANSI C code to configuration contexts as efficiently as possible, with as little modification to the source code as possible. Furthermore, the design cycle must be fast, to allow a high design iteration rate. To make this possible, a high-speed simulator is needed. The work was intended to be as generic as possible, to apply to as wide a range of architectures as possible—i.e. the tool chain must be *re-targetable*. However, it was necessary to bound this in some way: the work targets a family of dynamically reconfigurable arrays with the following properties:

- The reconfigurable array is in control of its own reconfiguration.
- Coarse-grained, where each functional unit supports operations similar to those in a typical RISC instruction set.

These two properties make the array sufficiently comparable to a microprocessor, so that a conventional C compiler can target them: being in control of its own reconfiguration allows the array to perform control flow; functional units supporting RISC-like instructions allow matching expressions to be written in a compiler back-end. Furthermore, coarse-grained architectures have a small configuration size, which means that they can be reconfigured much more rapidly than finer grained data path machines (such as FPGAs).

These properties mean that the work presented here is best suited to a single family of dynamically reconfigurable array: the reconfigurable instruction cell array (RICA [5]). The concepts could be easily applied to a wider range of architectures. However, this family of architectures represents a significant design space: the number of cells, the way they are interconnected, and the specific functionality of each cell type, is entirely open to exploration using these tools.

The sections of this chapter correspond to each of the main chapters in this thesis: emulation, scheduling, and pipelining. They begin by restating the problem description, aims and objectives that were made in the corresponding chapter, then show how the theories and results presented in those chapters satisfy these goals. The chapter concludes with an overall assessment of what the work achieves.

6.1 Emulation

6.1.1 Emulation: Problem Description

In order to validate application code and explore the design space, a high-speed simulation of the target architecture must be available.

Existing software-based methods of simulation for reconfigurable computing architectures are event-driven, and incur a sizeable time penalty for every configuration context. For instruction cell architectures, which have to be reconfigured many millions of times per second, this overhead eclipses the actual work done by the modelled processing units.

Software-based emulators are high-speed simulations available for more conventional microprocessor architectures. However, these do not provide support for *operation chaining*, which makes them unsuitable for data path machines, such as those considered by this thesis.

6.1.1.1 Aims

- Provide a software simulator to allow the design search space to be explored within a reasonable time frame.
- Allow rapid application development and validation.

6.1.1.2 Objectives

Generic: It must be easy to describe the target architecture (e.g. resource counts, timing figures), and the simulation adapt accordingly.

Extensible: It must be easy to add new functionality (e.g. cell types), preferably using a high-level description.

Fast: The simulation should be as close to real-time as possible.

Accurate: The simulation should behave as much as possible like the target architecture at a given level of abstraction, and should give a reasonable estimate of the timing.

6.1.2 Emulation: Demonstrated Outcomes and Contribution to Knowledge

A novel approach was suggested to reduce the per-step execution overhead seen in other simulators for data path architectures. This involves moving the resolution of the dependencies in the data paths into a pre-processing stage, prior to execution. When applied to an example processor, the results (section 3.4 on page 45) show that the execution speed achieved using this new approach is around two orders of magnitude higher than an equivalent SystemC model, and largely matches the speed of an FPGA model of the target reconfigurable instruction cell array. For the examples shown, this corresponds to a few percent of real-time performance.

This level of performance makes the proposed emulator suitable for use in feedback-directed optimisation, and thus could be an important part of future toolchains. This therefore satisfies the objective of execution speed. Note however that the performance decreases linearly with array size and utilisation. Real-time simulation of sizeable arrays with high-utilisation configuration contexts is unfeasible on any simulation running on a microprocessor, as the potential throughput of such arrays substantially exceeds the theoretical throughput of any microprocessor.

The emulator is highly adaptable to different types of reconfigurable processors with different functionality but similar control concepts, making it a good candidate for use in retargetable toolchains for hardware/software co-design. The cell count and mix can be freely modified at run-time,[1] making the simulation quite generic. The functionality is extensible by allowing new types of instruction cell or memory-mapped hardware to be expressed at a high-level using C++. These become available after recompiling the emulator.

The accuracy of the proposed simulation can be broken down into two components: functional accuracy, and timing accuracy. The functional accuracy is a function of how well the C++ description of each cell type matches the behaviour of the real cell. The serialisation algorithm ensures that the data flow occurs in an analogous manner to that in the real data paths in the target architecture. This guarantees that the state of the simulation matches that of the real system at the transition between each configuration context iteration. This is the same level of state coherence inherent in the design of the target architecture.

Similarly, timing accuracy comes by design—each configuration context specifies how many master clock periods it should persist for. The step loading time can also be known in advance. However, some hardware such as the shared data memory access arbiters introduce a dynamic delay, depending on contention whilst executing. Such data-path dependent delays cannot efficiently be modelled, without sacrificing significant performance. This doesn't turn out to be much of a problem, as internally pipelined cells are a more efficient way to deal with memory dependencies, so arrays supporting these typically do not incur dynamic delays. Note that the execution rate of the simulation is not a constant fraction of the real-life execution time on the target architecture.

6.1.3 Emulation: Further Work

The serialisation algorithm could be applied directly to translation, allowing even faster emulations to be performed. This should narrow the gap between the execution time of the emulation v.s. the original application code compiled natively for the microprocessor. This would be achieved by using the output of the serialisation algorithm to generate static call lists for each configuration context, which are then fed into an optimising linker (such as LLVM [60]) to generate an optimised, native binary, eliminating the overhead of interpretation [99]. Note however that a performance gap may still exist, due to the relative maturity of the optimisations available in the host compiler v.s. that of the target reconfigurable architecture.

[1] prior to loading the configuration contexts.

For more complex cell types, where some inputs of a given cell are combinatorial whilst other inputs of the same cell are synchronous, the concept of *disjoint* cells no longer applies. The serialisation algorithm in its current form is unable to deal with such cell types. This problem can be resolved by making the *disjoint* property apply to each input individually (instead of the output), and altering the serialisation algorithm accordingly. The serialisation algorithm described in this thesis determines when a given *operate* action is ready to be added by looking at the output of each operation feeding the inputs of the current operation. This would be modified by serialising in reverse, allowing each input to be considered independently. Combinatorial inputs would be sampled in the *operate* action, and synchronous (*disjoint*) inputs would be sampled in the *update* action. Note that this requires that inputs can be sampled in the *update* action. The *update* actions are serialised in arbitrary order, so to avoid corruption of the data paths, it is important that the cell outputs are not modified in any *update* action. This imposes a new design rule when implementing new instruction cell types.

Multi-core: Due to the execution rate being dependent on the data path complexity, each configuration context may execute at a different fraction of real-time performance. As a result, the simulation of multiple cores running different programs will not be able to consistently run at maximum possible execution rate, without losing synchronisation with the other cores.[2] This means that performance doesn't necessarily scale well with the number of cores. However, measures can be taken to minimise this loss of performance, by only synchronising the threads at changes in state that are visible to the other cores.[3]

[2] when each core is executed as a separate thread.
[3] e.g. memory access conflicts/race conditions.

6.2 Scheduling

6.2.1 Scheduling: Problem Description

A complete toolchain for allowing a dynamically reconfigurable architecture to be programmed from C must convert C source code into a set of configuration contexts for the target architecture. The approach used here was to leverage existing re-targetable compiler technology to compile the C code into an intermediate representation, which a separate tool—the *scheduler*—then converts into configuration contexts.

The intermediate representation chosen here was a RISC-like assembly, consisting of instructions grouped into basic blocks. Each instruction matches the functionality of one of the cell types in the target architecture. Instruction operands are registers, which correspond to physical registers in the target architecture.

Working from assembly has the benefit of allowing conventional re-targetable compiler technology to be leveraged with little modification. However, this approach produces a serialisation with very low parallel efficiency (suffering from the ILP wall [33]), making little use of the available resources in a reconfigurable computing architecture. Parallelisation is performed by the stand-alone scheduler tool.

Parallelisation involves reconstructing the data flow graph (DFG) from the assembly instructions of each basic block, replacing the internal uses of registers with wires. Some of these wires become registers again, if that connection has to span the boundary between different configuration contexts, which will happen if there are insufficient resources to map all instructions into a single configuration context. This mapping is performed by a scheduling algorithm.

Another down-side of working from the assembly is that high-level information is not available—all information must be extracted from just the instructions. The assembly is only able to express when a register receives a new value; it is not able to express when its current value becomes irrelevant. The compiler's use of registers to pass information between the operands of instructions inside a basic block mean that an excessive number of registers are written to.

Parallelisation on a resource constrained core requires splitting data paths over multiple configuration contexts, which requires an increased number of registers to store the values of the connections that span each boundary. This increase is due to the fact that several of the broken connections will correspond to the same register in the assembly, but at different times.[4] These must be assigned a separate physical register each. Register starvation would normally lead to a failure to parallelise. In order to avoid register starvation, it is important to determine which registers really store important information.

[4] thus representing a different piece of information.

6.2.1.1 Aims

Correctness: The configuration contexts for the target architecture must exhibit the same external state change behaviour as the original code (assembly).

Efficiency: The total execution time of the configuration contexts should be as low as possible.

6.2.1.2 Objectives

- Devise a data model that can describe a wide range of target architectures, in a manner that allows for easy static analysis.
- Devise a series of algorithms that operate on this data model, to efficiently transform basic blocks into valid configuration contexts.

Most of the work presented on the scheduler focuses on maximising this mapping efficiency, in terms of number of configuration contexts produced, total critical path, throughput, and register activity.

6.2.1.3 Novelty

A form of list scheduling was devised that focussed on packing data paths into as few steps as possible. The side effect of this packing is to split more data paths across step boundaries, which increases the demand for registers. For cores with a very limited number of registers, this can lead to register starvation, which reduces parallel efficiency. A series of algorithms were devised to avoid this (register starvation avoidance, section 4.10), with minimal impact on the efficiency of the resulting schedule.

Furthermore, a series of optimisation and analysis passes were presented that improve the scheduling efficiency (live register identification, section 4.7), and aid the routing tool to achieve a more optimum allocation over the whole program (global register reallocation, section 4.12). This improves routability and reduces combinatorial delay, thus improving throughput.

6.2.2 Scheduling: Demonstrated Outcomes and Contribution to Knowledge

The results for applying the proposed tree follower scheduling algorithm (section 4.9 on page 87) to examples that are constrained in terms of computation resources (section 4.13.1 on page 117) show that the scheduling algorithm was able to always meet the theoretical minimum step count, as was the design intent. However, despite minimising the step count, the tree follower scheduling algorithm can produce sub-optimal schedules in terms of total critical path[5]. This is due to the visitation order: when a particular resource is constrained, the choice of which uses of that resource are scheduled in each step should depend on where these operations occur in terms of position in the original data flow graph. To minimise total critical path, the operations that map to the constrained resource type should be scheduled in ascending order of position in the original data flow graph.

[5] the sum of the critical paths of each step produced.

The tree follower does not ensure this visitation order—instead, it prefers to continue up a data path arm, rather than move to others with similar geometry. If the arm contains more than one operation of the constrained type in sequence, then precedence is given to these dependent operations, instead of to the independent operations in other arms. The arms end up with less overlap in the resulting schedule, which extends the critical path. A mobility-based scheduling algorithm should exhibit different behaviour in this regard, preferring to schedule the operations in order of their overall position in the data flow graph. Mobility-based list scheduling algorithms are less computationally intensive than the tree follower algorithm proposed here.

Ideally the scheduling algorithm would have been compared to existing alternatives such as mobility-based list scheduling[57], however this was not possible within the available time, as the code base was built around the tree follower concept, and would need significant alteration. The tree follower was originally intended to work around certain hardware constraints that were difficult to deal with through list scheduling; however, these were later removed. The hardware had evolved a lot since the previous published work on list scheduling, and the tool created from that work was tailored towards a particular hardware design and selection of cell types. This made the two tools impossible to compare directly. This would be a useful focus for some future work.

Section 4.7 on page 75 proposed an algorithm for determining which registers named in the assembly contain important data across the boundaries between basic blocks, by examining the program control flow graph (CFG). The CFG is reconstructed from the assembly using another algorithm proposed in section 4.6 on page 72. The effect of this is demonstrated in section 4.13.2 on page 122, where it can be seen to increase the available register pool by many times. More importantly, the advantage increases with core size (i.e. total number of registers). For a given core size, this means that fewer (if any) registers need to be reserved for scratch, which gives the compiler more room to produce larger basic blocks, which in turn are easier to parallelise. It also frees up registers for use in assembly-level optimisations, such as the conversion of stack-local variables (memory accesses) to registers.

The processing time and memory requirements of the live register identification algorithm scale exponentially with program complexity. Image signal processing applications like the examples in sections 4.13 and 5.8 take a few seconds to process, whilst programs of the size of the H.264 decoder can take a minute or more. This is acceptable for the domains targeted by the target architecture, but does not bode well for desktop applications. However, this can be addressed by breaking the program up into separate units that can be analysed individually (e.g. at the function level or compilation unit level), which makes the processing time scale linearly with the number of units. The boundaries between these units would be treated in the same manner as if live register identification had not been done. This compromises the extent to which registers are made available, but overall will still lead to a significant improvement compared to not performing live register identification at all.

In a core that is both computation resource-starved and register-starved, the scheduling algorithm was shown in section 4.13.3 on page 125 to produce a valid schedule in all cases, with varying use of the three implemented forms of register starvation avoidance ('rewind', 'shuffle', and 'split'—section 4.10 on page 95). Even in the very extreme cases tested—with a 50% reduction in available registers—the total critical path increased by a maximum of 17%, leading to an 11% reduction in throughput. This combination of technologies makes it possible for the scheduling algorithm to achieve significantly increased parallelism even in very highly constrained cores, avoiding the need to revert to using the serialisation given in the assembly.

As the core size increases, the allocation of cells to physical resources becomes increasingly significant, as the maximum possible distance between two arbitrarily placed cells increases. Reallocation can be used to bring the end points of connections closer together. However cells that maintain internal state can only be reallocated globally. Section 4.12 on page 109 presented an algorithm for tracking information being passed between the steps of a program via registers. This knowledge can be used to decouple registers between steps, allowing them to be reallocated more freely. One particular use of this—called *register renaming*—was demonstrated in section 4.13.4 on page 130 on a large core, resulting in a 30% reduction in average path length and interconnect usage. This should allow more complex steps to be routed, and improves the performance of simpler steps, by reducing the critical path.

6.2.3 Scheduling: Further Work

Improved scheduling algorithm: Implement a mobility-based scheduling algorithm, using the register starvation avoidance methods described in this thesis. The rewind method would alter the definition of the ready list according to previous attempts. The shuffle method would rearrange the order of operations of equal standing in the ready list.

Scheduling of data: Automating the mapping of variables and arrays to registers and non-uniform memory such as stream buffers and embedded register files [100]. This will require changes to the compiler, since such information is not available from the assembly.

Alternative intermediate representation: Many of the issues of working from assembly can be avoided by moving to higher-level internal representations inside the compiler, e.g. TreeSSA / GIMPLE [101, 102] in GCC, or the LLVM intermediate representation [60].

Place and route: The output of the scheduler tool proposed in this work is in the form of an *abstract netlist*; it describes only which cells are connected together, not how they are connected[6]. This mapping is the task of a separate place and route (*mapper*) tool, which is outside the scope of this thesis. Such a tool has been created [61].

Hardening: The abstract netlist can alternatively be used to derive a static connectivity map (look-up table) for each active cell in the array, allowing a *hardened* core to be produced. This avoids the area and combinatorial delay overhead of the reconfigurable interconnect, at the expense of flexibility—the resulting array is only able to execute the single netlist (program) that it was produced for. This provides an alternative C-to-gates tool flow, with an unusually high degree of silicon re-use.

[6] i.e. what path they take through the reconfigurable interconnect.

6.3 Pipelining

6.3.1 Pipelining: Problem Description

The high degree of operation chaining available in dynamically reconfigurable arrays has the advantage of being able to break through the *ILP wall* [33], by allowing dependant operations to be connected together via *wires*. This largely avoids the central register file common in microprocessors and their derivatives, and thus avoids the bandwidth constraints imposed by it. The down-side to chaining large sequences of dependent operations together is that the combinatorial delay of the critical path increases, thus hurting throughput—particularly in single configuration context loops (*kernels*), which are the most efficient way to perform heavy computation.

Each instruction cell is idle until the output of each instruction cell on which it depends settles, and is idle again once its output has settled. It remains idle until the next context iteration begins. The longer the chain of dependent operations, the higher the fraction of execution time the cell is idle for. Pipelining these chains of dependent operations allows the critical path to be reduced, thus increasing the iteration rate and thus the throughput. The situation is complicated by the fact that each instruction cell type has a different combinatorial delay, as do the interconnect paths.

6.3.1.1 Aims

- Automatically pipeline compute-intensive loops to significantly increase throughput.

6.3.1.2 Objectives

- Automatic pipeline stage assignment, based on a user-supplied target critical path constraint.
- Minimal hardware changes.
- Minimising the impact of pipelining on the context configuration size.
- Minimising the impact of pipelining on the overall program size.
- Automating the choice of target critical path.

6.3.2 Pipelining: Demonstrated Outcomes and Contribution to Knowledge

Structural-level pipelining techniques were applied via software to rapidly reconfigurable / programmable architectures supporting operation-chaining, where complete kernels are mapped into a single configuration context/cycle. This improves throughput by reducing the critical path length of the looping kernel.

Furthermore, this work introduced the novel idea of achieving pipeline filling and flushing through dynamic reconfiguration (multi-step pipelining), in a manner similar to that used in software pipelining. This has the effect of simplifying the design of the pipelined kernel, removing the need for additional control logic to be added to initialise the pipeline, thus requiring no changes to the existing hardware. The addition of pipelining in this manner was shown to increase the register requirement, and uses more program memory to store the additional configuration contexts (prologue and epilogue). This approach however was found not to be very scalable, as the overhead in terms of additional configuration contexts would quickly dominate the program size. This makes multi-step pipelining only suitable to relatively shallow pipelines (e.g. up to about 10 stages), which is appropriate for small cores with up to a few hundred cells.

An improvement on this (single-step pipelining), requiring some hardware changes, was investigated. With additional constraints imposed on the pipeline stage assignment, it is possible to strip away most of the additional configuration information, leaving a single configuration context that performs all the pipeline execution phases: fill, loop, and flush. The only overhead is a few bits of configuration data per configuration context, specifying the number of pipeline stages that exist in that step. The hardware overhead is very small—an additional counter (pipeline depth counter), plus a 2-bit state signal (execution phase) broadcast to the instruction cells in the array that maintain state (except for registers). This typically represents a very small fraction of the cells in the core. The novelty in this approach lies in the algorithmic work-arounds that alter the pipeline geometry, in order to allow deep pipelining to be possible with such minimal hardware additions. The limitation of this approach is an increase in pipeline stage registers required to work around the changes in the pipeline geometry (i.e. operations with side effects must be placed in the first or last pipeline stage, and extra cells are sometimes needed to supply the initial value of certain kernel registers). The cell mix therefore has to be chosen to ensure sufficient register availability for the given core size. By placing registers in the interconnect instead of in cells, single-step pipelining becomes inherently scalable, as registers will always be available along each connection in the data flow graph when mapped to the array, and their availability scales with the length of the connection (which itself largely determines the number of pipeline stages needed along the connection).

In both approaches, the potential throughput is limited by the number of registers available for use in connecting the pipeline stages, and by the presence of feedback loops that demand single cycle latency (e.g. when updating the value of a register).

The pipelining algorithms were applied to a simple demosaic filter for a variety of target throughput constraints, and achieved a maximum throughput of nearly ten times that of the original kernel. The technique was shown to also improve the throughput for applications where the kernel performs only a small number of iterations—particularly when using the single-step pipelining approach, which eliminates the step loading time for each fill and flush step. A two pass 8x8 2-D DCT was used to demonstrate this: the 8 element 1-D DCT kernel performs only 8 iterations in each pass (rows and columns), and yet a speed-up of 35% was shown to be possible (section 5.8.1 on page 165).

In general, pipelining is able to take a kernel of arbitrary critical path, and reduce it to the minimum critical path determined by non-pipelineable data paths (which is a constant for a given core). The maximum achievable speed-up therefore scales with kernel complexity, so long as sufficient registers are available, and so long as the kernel performs sufficient iterations over which to amortise the impact of the additional fill and flush iterations.

The pipelining algorithm also allows for the use of internally pipelined cells, which are useful for reducing the critical path of a pipelined kernel, and for making combinatorial operations synchronous. A common use of this is to hide memory latency. This was demonstrated with a gamma correction module which performs table look-ups requiring very high memory bandwidth. Pipelining was shown to increase the throughput by up to nearly six times (42MPixels/s → 242MPixels/s) using combinatorial memory reads, and nine times (42MPixels/s → 379MPixels/s) using internally pipelined memory reads (section 5.8.2 on page 174).

This work concentrated on reconfigurable instruction cell processors, which support a high degree of operation chaining. However, the same techniques could be applied to other quite different architectures that support operation chaining, such as upcoming VLIW/ULIW processors [31].

Furthermore, this work proposed an algorithm for automatically applying dynamic structural-level pipelining to single configuration context kernels running on dynamically reconfigurable arrays (DRAs). The technique is a form of feedback directed optimisation, where profiling information (consecutive execution counts) are used to determine which kernels will benefit from pipelining. Candidates with very low consecutive execution counts must not be pipelined too deeply. This is to ensure that the additional latency of pipeline filling and flushing is more than nullified by the decrease in total execution time for the pipelined kernel loop when the pipeline is full. This is only possible when the minimum possible iteration count is known. This is the case for pixel-level kernels in the ISP application domain, as the iteration count is typically the line size of the image.

An iterative approach is used to form an efficient pipeline, where the timing constraint is automatically chosen to be an integer multiple of the master clock frequency. The timing constraint is incremented until a valid pipeline can be constructed without encountering register starvation. The range of possible pipeline geometries is controlled by the availability of registers. Architectures with distributed registers will offer the best results, otherwise the bandwidth of the interface and/or additional combinatorial delays introduced by routing to and from a register file would likely outweigh any benefit. This makes the case for registers to be made available in the interconnect itself.

The algorithm was applied to a demosaic kernel of modest complexity and to a software division algorithm, leading to the possibility to pipeline to a significant depth. A performance increase of up to 7 times can be obtained for the demosaic example, and nearly 10 times for the division (section 5.8.3 on page 179). As the pipeline gets deeper, the cost—in terms of register requirement and storage for pipeline filling and flushing contexts—increases more than linearly. As the critical path of the pipelined kernel gets smaller, the quantisation of the iteration rate caused by the master clock, gets increasingly worse. Inside the bounds of this quantisation, reducing the pipeline critical path[7] has no effect on the iteration rate. In these situations, extra

[7] by increasing the number of pipeline stages.

resources would be introduced for no benefit. To avoid this, the proposed algorithm relaxes the critical path to take into account this quantisation, thus minimising the resource requirements for a given physically achievable iteration rate.

6.3.3 Pipelining: Further Work

Re-timing: To allow infinite impulse response filters to be pipelined (to some extent). Registers that form the feedback loop(s) must be re-used as pipeline stage registers. This could be modelled by adding constraints between operations on either side of a feedback register forcing them to be one pipeline stage apart, then modifying the pipeline stage register assignment pass to use the existing registers. This is similar to the modifications that were added to support internally pipelined cells.

Post-routing pipelining: As with the rest of the work presented in this thesis, pipelining is performed on abstract netlists, where the path length information is not yet known. As a result, pipelining is performed prior to routing, based on a uniform path length estimate. Although this certainly leads to higher real-life throughputs, it may not be optimal, due to variance in the path lengths. This problem increases with array size. To compensate for this, large arrays are proposed to have registers available in the interconnect (sboxes). When this is available, the alternative approach is to perform pipelining after routing. This involves using the existing pipeline stage assignment algorithm but using the real path length information. Pipeline stage register assignment is different: the pipeline stage information is used to determine how many delay elements (registers) to enable along each path, and a new algorithm decides which registers along that path should be enabled, in order to best meet the target critical path length. Hints must be provided to the routing tool to ensure that certain paths are long enough (e.g. those leading to output registers), and others (e.g. non-pipelineable data paths) are as short as possible. For best results, multiple routing/pipelining iterations might be necessary. This has been investigated and implemented, but was not presented in this thesis.

6.4 Closing Remarks

The work presented in this thesis has been incorporated into a commercial tool set for the RICA architecture. The combination of proposed tools and algorithms provide a high degree of design automation—going from high-level C code to a given target architecture. Design space exploration currently remains a manual task, but the flexibility of the tools makes this quite easy. Cores of varying sizes and resource mix can be tested with just a change in a text file—Machine Description File (MDF)—describing those parameters. A resulting configuration and performance metrics can be obtained by simply re-running the tools. The addition of new functionality—e.g. new cell types or hardware—is more involving, requiring that the compiler be modified to support the new instruction, and the emulator be modified to support the functionality of that new resource.

The design premise of going from C to reconfigurable hardware is to allow easier programming of high-throughput architectures. The pure von Neuman programming model—where algorithms are expressed imperatively, operating on a single shared data memory—offers the simplest way of describing algorithms. However, to get maximum performance on large arrays, some degree of customisation of the C code is required, straying away from this model. This is because the current compiler is unable to deal with non-uniform memory architectures (NUMA). The scheduler is currently able to work around this to some extent, by converting stack local variables to registers, where possible. However, without high-level information defining how the various variables and arrays are used in a program, the scheduler is unable to perform more complex transforms such as mapping arrays to stream buffers. There is encouraging research in these directions however [103], especially within the LLVM community [104], so this should be possible to provide in the near future. However, there is only so much time available to do a PhD.

Appendix A
Emulator Test Programs

```
/*
 * Parallel.c
 *
 * Created by Mark Muir on 2008-06-24
 *
 * Example program with a single-step kernel demonstrating several independent
 * chains of dependent operations running in parallel. This is to aid in the
 * comparison between the execution times of the simulator and emulator.
 */

#define DATASET_SIZE    200
#define REPEAT          500

int input [DATASET_SIZE];
int output[DATASET_SIZE];

int main()
{
    int i, j;

    for (i=0; i<DATASET_SIZE; i++)
        input[i] = i;

    for (j=0; j<REPEAT; j++)
    {
        /* Perform a set of operations in parallel. */
        for (i=0; i<DATASET_SIZE; i+=4)
        {
            int result[4];
            result[0] = (input[i+0] & i) * (i<<1);
            result[1] = (input[i+1] & i) * (i<<2);
            result[2] = (input[i+2] & i) * (i<<3);
            result[3] = (input[i+3] & i) * (i<<4);
            /* Write the results after reading all values, to avoid memory
               access conflicts. */
            output[i+0] = result[0];
            output[i+1] = result[1];
            output[i+2] = result[2];
            output[i+3] = result[3];
        }
    }

    return 0;
}
```

Figure A.1: C source code for the example with four copies of the data path executing independently, in parallel.

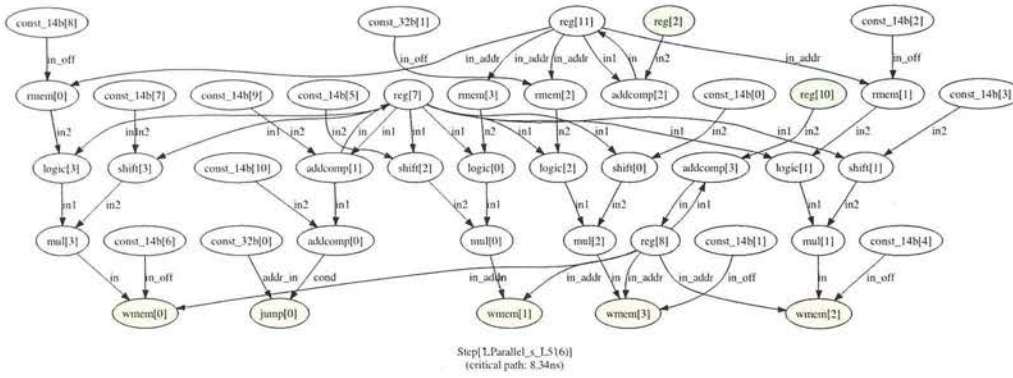


Figure A.2: Data flow graph for the configuration context corresponding to the main loop (kernel) of the 'parallel' example program shown in figure A.1. Generated by the RICA tools.

```
/*
 * Combinatorial.c
 *
 * Created by Mark Muir on 2008-06-24
 *
 * Example program with a single-step kernel demonstrating long chains of
 * dependent operations. This is to aid in the comparison between the
 * execution times of the simulator and emulator.
 */

#define DATASET_SIZE    200
#define REPEAT          500

int input [DATASET_SIZE];
int output[DATASET_SIZE];

int main()
{
    int i, j;

    for (i=0; i<DATASET_SIZE; i++)
        input[i] = i;

    for (j=0; j<REPEAT; j++)
    {
        /* Perform a set of operations with similar cell resource
           requirements, in series. To keep the memory activity (WMEM
           count) the same as the Parallel example, the same value is
           written to four consecutive addresses. */
        for (i=0; i<DATASET_SIZE; i+=4)
        {
            int result[4];
            result[0] = ((input[i+0] & i) << i);
            result[1] = ((input[i+1] & i) << i);
            result[2] = ((input[i+2] & i) << i);
            result[3] = ((input[i+3] & i) << i);
            /* Pass the temporary results through memory, to prevent
               the compiler from re-ordering the multiplication a*b*c*d
               to (a*b)*(c*d) instead of the desired ((a*b)*c)*d. */
            output[i+0] = result[0] * i;
            output[i+1] = result[1] * output[i+0];
            output[i+2] = result[2] * output[i+1];
            output[i+3] = result[3] * output[i+2];
        }
    }

    return 0;
}
```

Figure A.3: C source code for the example with two copies of the data path executing independently in parallel, with another two copies of the data path dependent on these (thus extending the critical path).

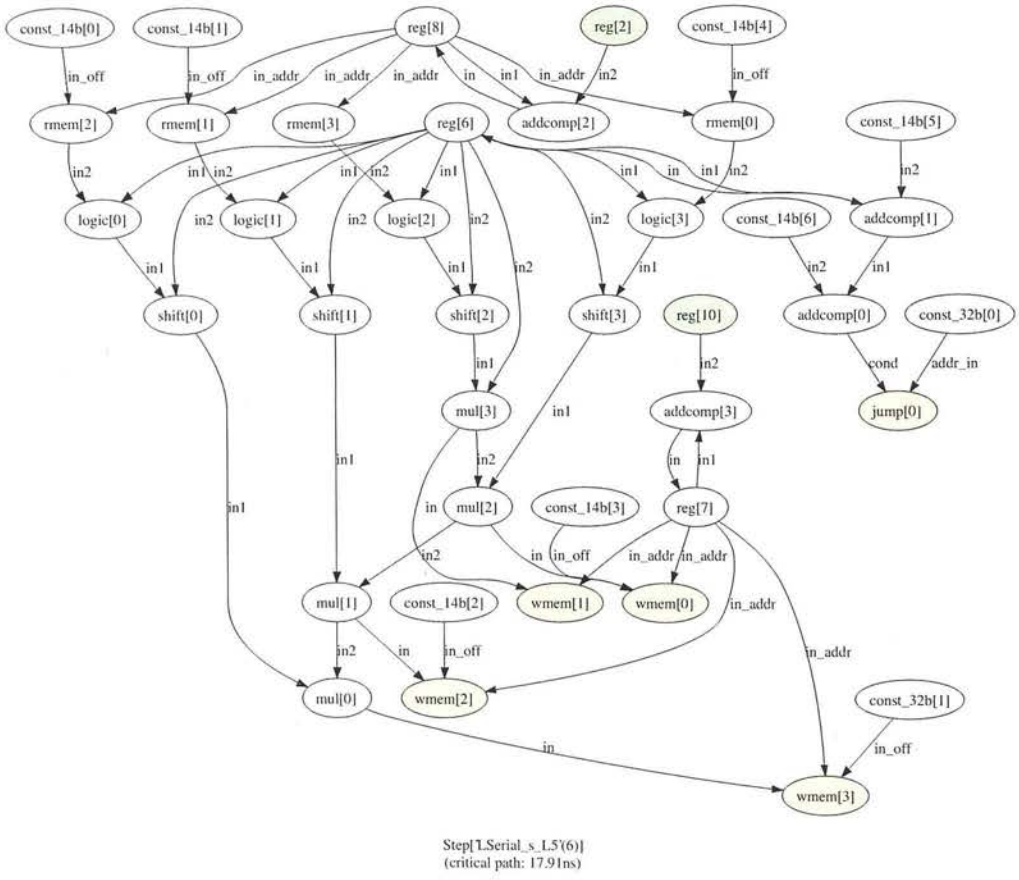


Figure A.4: Data flow graph for the configuration context corresponding to the main loop (kernel) of the 'combinatorial' example program shown in figure A.3. Generated by the RICA tools.

```
/*
 * Sequential.c
 *
 * Created by Mark Muir on 2008-06-24
 *
 * Example program with a single-step kernel demonstrating the same independent
 * chains of dependent operations as the Parallel example, but with the
 * independent chains placed in different steps (by being in different loop
 * iterations). This is to aid in the comparison between the execution times of
 * the simulator and emulator.
 */

#define DATASET_SIZE    200
#define REPEAT          500

int input [DATASET_SIZE];
int output[DATASET_SIZE];

int main()
{
    int i, j;

    for (i=0; i<DATASET_SIZE; i++)
        input[i] = i;

    for (j=0; j<REPEAT; j++)
    {
        /* Perform an arbitrary operation on each member of the data set. The
           resuls of the operation will be different to the Parallel example,
           since the value of 'i' will be different in some iterations.
           Compensating for this would change the complexity of the operation. */
        for (i=0; i<DATASET_SIZE; i++)
        {
            output[i] = (input[i] & i) * (i<<1);
        }
    }

    return 0;
}
```

Figure A.5: C source code for the example with the data path executed inside a loop (which hasn't been unrolled), causing the main loop to consist of four iterations of the same configuration context executing in sequence.

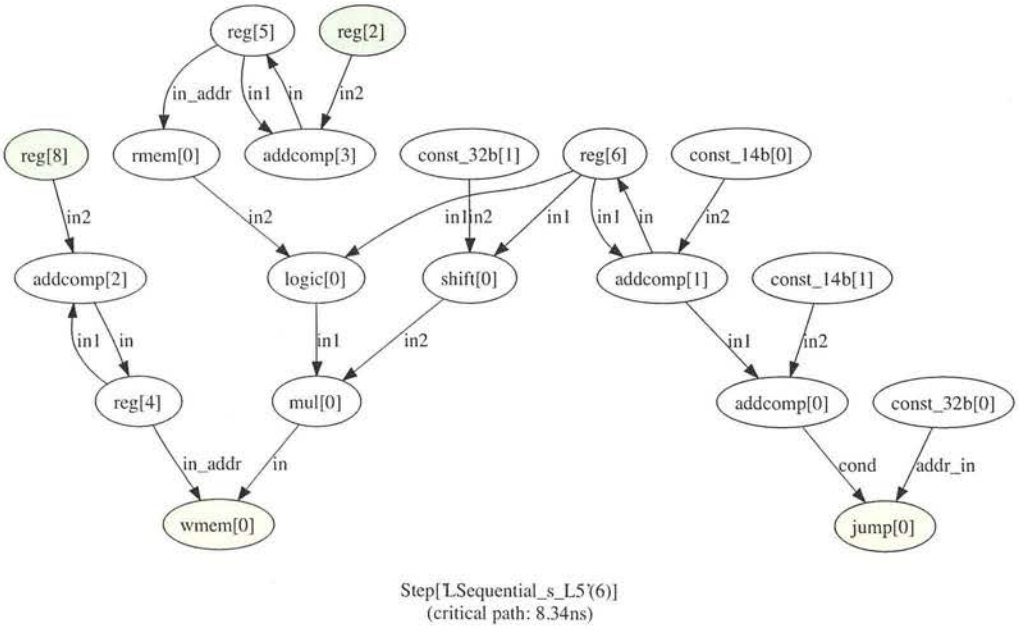


Figure A.6: Data flow graph for the configuration context corresponding to the main loop (kernel) of the 'sequential' example program shown in figure A.5. Generated by the RICA tools.

Appendix B
Live Register Identification Algorithm Trace

CFG edge	Action	Registers live on exit from caller
NULL → _reset	visit _main	N/A
_reset → _main	visit L1	-
_main → L1	visit _func	-
L1 → _func	visit L4	-
_func → L4	visit L4	-
L4 → L4	visit L4	-
L4 → L4	already considered	- + **r3,r4,r6** + -
	⇒ update and return	= r3, r4, r6
L4 → L4	visit ret2	r3, r4, r6
L4 → ret2	visit L2	r3, r4, r6
ret2 → L2	visit _func	-
L2 → _func	visit L4	-
_func → L4	already considered	- + **r3,r4,r6** + **r3,r4,r6**
	⇒ update and return	= r3, r4, r6
L2 → _func	all targets visited	- + **r1,r2,r6** + **r3,r4,r6**
	⇒ update and return	= r1, r2, r3, r4, r6
ret2 → L2	all targets visited	- + **r5,r11** + **r1,r2,r6**
	⇒ update and return	= r1, r2, r5, r6, r11
L4 → ret2	visit L3	r3, r4, r6
ret2 → L3	visit ret1	r5, r11
L3 → ret1	visit _end	-
ret1 → _end	no targets	- + - + -
	⇒ update and return	= -
L3 → ret1	all targets visited	- + **r1,r9** + -
	⇒ update and return	= r1, r9
ret2 → L3	visit L1	r5, r11
L3 → L1	visit _func	r1, r9
L1 → _func	already considered	- + **r1,r2,r6** + **r3,r4,r6**
	⇒ update and return	= r1, r2, r3, r4, r6
L3 → L1	all targets visited	r1, r9 + **r5** + r1,**r2,r6**
	⇒ update and return	= r1, r2, r5, r6, r9
ret2 → L3	all targets visited	r5, r11 + r5,r11 + **r1,r2**,r5,**r9**
	⇒ update and return	= r1, r2, r5, r9, r11
L4 → ret2	all targets visited	r3, r4, r6 + **r1**,r3,**r9** + **r1,r5,r9**
	⇒ update and return	= r1, r3, r4, r5, r6, r9
L4 → L4	all targets visited	r1, r3, r4, r5, r6, r9 + r3,r4,r6 + r1,r3,r4,r5,r6,r9
	⇒ update and return	= r1, r3, r4, r5, r6, r9
_func → L4	all targets visited	r3, r4, r6 + r3,r4,r6 + **r1**,r3,r4,**r5**,r6,**r9**
	⇒ update and return	= r1, r3, r4, r5, r6, r9
L1 → _func	all targets visited	r1, r2, r3, r4, r6 + r1,r2,r6 + r1,r3,r4,**r5**,r6,**r9**
	⇒ update and return	= r1, r2, r3, r4, r5, r6, r9
_main → L1	all targets visited	- + **r5** + **r1,r2,r5,r6**
	⇒ update and return	= r1, r2, r5, r6
_reset → _main	all targets visited	- + **r1,r2,r9** + **r1,r2,r6**
	⇒ update and return	= r1, r2, r6, r9
NULL → _reset	all targets visited	N/A
	⇒ update and return	
information changed ⇒ traverse again		

Continued in table B.2 ...

Table B.1: Trace of the CFG walk for the example in figure 4.18 on page 77, using information in table 4.4 on page 79. The 'update' of the record of registers live on exit from the caller (LHS) consists of adding all the input registers of the callee (RHS), plus any registers live on exit from the callee (RHS) that weren't clobbered in the callee (RHS). Newly added registers resulting from the update are shown in bold.

CFG edge	Action	Registers live on exit from caller
	... Continued from table B.1	
NULL → _reset	visit _main	N/A
_reset → _main	visit L1	r1, r2, r6, r9
_main → L1	visit _func	r1, r2, r5, r6
L1 → _func	visit L4	r1, r2, r3, r4, r5, r6, r9
_func → L4	visit L4	r1, r3, r4, r5, r6, r9
L4 → L4	visit L4	r1, r3, r4, r5, r6, r9
L4 → L4	already considered	r1, r3, r4, r5, r6, r9 + r3,r4,r6 + r1,r4,r5,r9
	⇒ update and return	= r1, r3, r4, r5, r6, r9
L4 → L4	visit ret2	r1, r3, r4, r5, r6, r9
L4 → ret2	visit L2	r1, r3, r4, r5, r6, r9
ret2 → L2	visit _func	r1, r2, r5, r9, r11
L2 → _func	visit L4	r1, r2, r3, r4, r6
_func → L4	already considered	r1, r3, r4, r5, r6, r9 + r3,r4,r6 + r1,r3,r5,r6,r9
	⇒ update and return	= r1, r3, r4, r5, r6, r9
L2 → _func	all targets visited	r1, r2, r3, r4, r6 + r1,r2,r6 + r1,r3,r4,r5,r6,**r9**
	⇒ update and return	= r1, r2, r3, r4, r6, r9
ret2 → L2	all targets visited	r1, r2, r5, r9, r11 + r5,r11 + r1,r2,**r6**
	⇒ update and return	= r1, r2, r5, r6, r9, r11
L4 → ret2	visit L3	r1, r3, r4, r5, r6, r9
ret2 → L3	visit ret1	r1, r2, r5, r6, r9, r11
L3 → ret1	visit _end	r1, r2, r5, r6, r9
ret1 → _end	no targets	- + - + -
	⇒ update and return	= -
L3 → ret1	all targets visited	r1, r2, r5, r6, r9 + r1,r9 + -
	⇒ update and return	= r1, r2, r5, r6, r9
ret2 → L3	visit L1	r1, r2, r5, r6, r9, r11
L3 → L1	visit _func	r1, r2, r5, r6, r9
L1 → _func	already considered	r1, r2, r3, r4, r5, r6, r9 + r1,r2,r6 + r1,r3,r4,r5,r6,r9
	⇒ update and return	= r1, r2, r3, r4, r5, r6, r9
L3 → L1	all targets visited	r1, r2, r5, r6, r9 + r5 + r1,r2,r5,r6
	⇒ update and return	= r1, r2, r5, r6, r9
ret2 → L3	all targets visited	r1, r2, r5, r6, r9, r11 + r5,r11 + r1,r2,r5,r9
	⇒ update and return	= r1, r2, r5, r6, r9, r11
L4 → ret2	all targets visited	r1, r3, r4, r5, r6, r9 + r1,r3,r9 + r1,r5,r9
	⇒ update and return	= r1, r3, r4, r5, r6, r9
L4 → L4	all targets visited	r1, r3, r4, r5, r6, r9 + r3,r4,r6 + r1,r3,r4,r5,r6,r9
	⇒ update and return	= r1, r3, r4, r5, r6, r9
_func → L4	all targets visited	r1, r3, r4, r5, r6, r9 + r3,r4,r6 + r1,r3,r4,r5,r6,r9
	⇒ update and return	= r1, r3, r4, r5, r6, r9
L1 → _func	all targets visited	r1, r2, r3, r4, r5, r6, r9 + r1,r2,r6 + r1,r3,r4,r5,r6,r9
	⇒ update and return	= r1, r2, r3, r4, r5, r6, r9
_main → L1	all targets visited	r1, r2, r5, r6 + r5 + r1,r2,r5,r6
	⇒ update and return	= r1, r2, r5, r6
_reset → _main	all targets visited	r1, r2, r6, r9 + r1,r2,r9 + r1,r2,r6
	⇒ update and return	= r1, r2, r6, r9
NULL → _reset	all targets visited	N/A
	⇒ update and return	
	information changed ⇒ traverse again	
	⋮	
	no new information ⇒ end	

Table B.2: Continuation of the trace of the CFG walk for the example in figure 4.18 on page 77.

Appendix C
Pipelining Test Programs

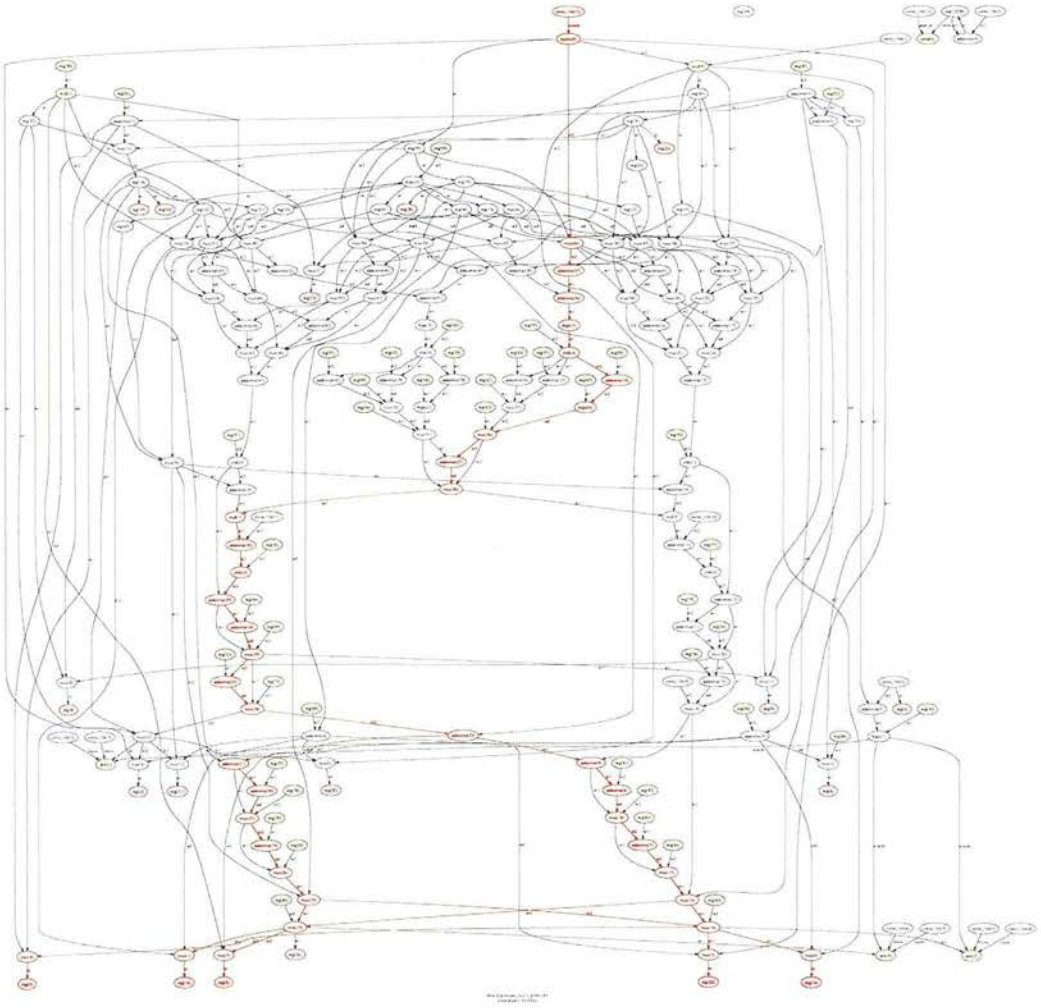


Figure C.1: Data flow graph for the configuration context corresponding to the main loop (kernel) of the 3x3 demosaic module used in section 5.8.1 on page 165. Generated by the RICA tools. Not pipelined. The critical path is shown by a red outline.

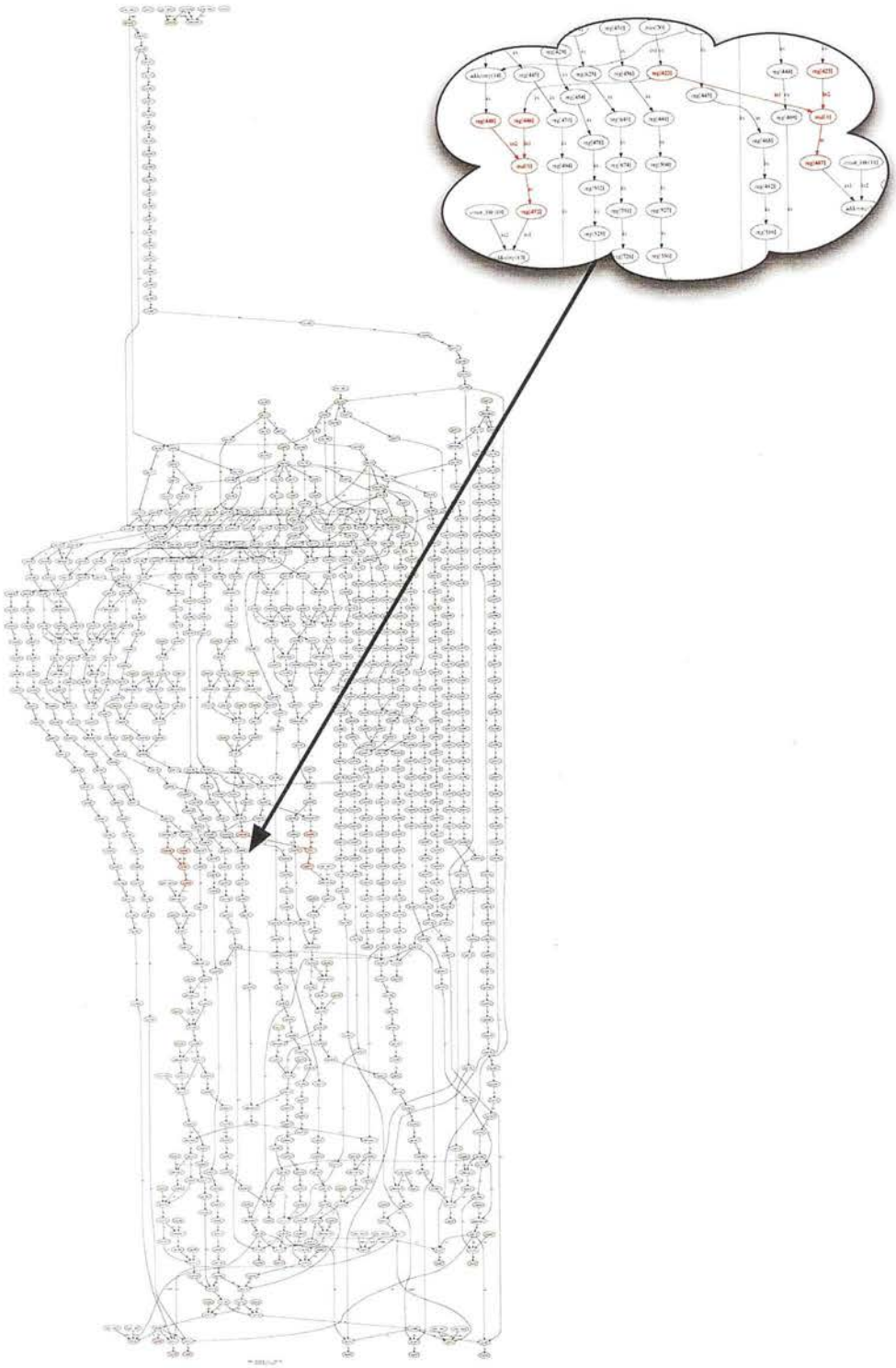


Figure C.2: Data flow graph for the configuration context corresponding to the main loop (kernel) of the 3x3 demosaic module used in section 5.8.1 on page 165. Generated by the RICA tools. Maximally pipelined (target 5ns, single-step pipelining). Pipeline stage registers are shown with a pink fill. The critical paths are shown by a red outline (clearer in the zoomed-in view). Two adjacent pipeline stages share the same critical path; both consist of a multiplier between pipeline stage registers.

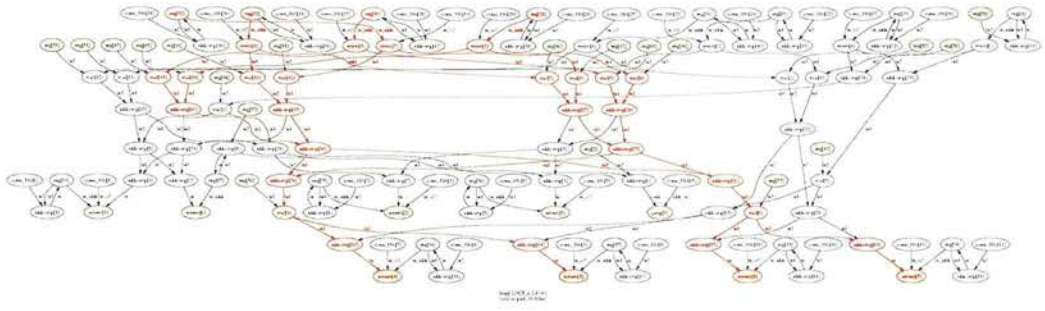


Figure C.3: Data flow graph for the configuration context corresponding to the main loop (kernel) of the DCT example used in section 5.8.1 on page 165. Generated by the RICA tools. Not pipelined. The critical path is shown by a red outline.

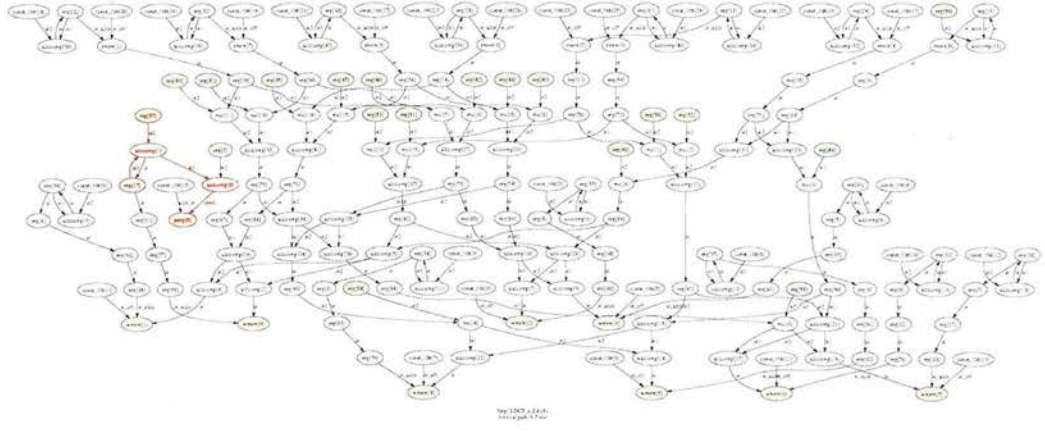


Figure C.4: Data flow graph for the configuration context corresponding to the main loop (kernel) of the DCT example used in section 5.8.1 on page 165. Generated by the RICA tools. Maximally pipelined (target 7ns, single-step pipelining). Pipeline stage registers are shown with a pink fill. The critical path is shown by a red outline.

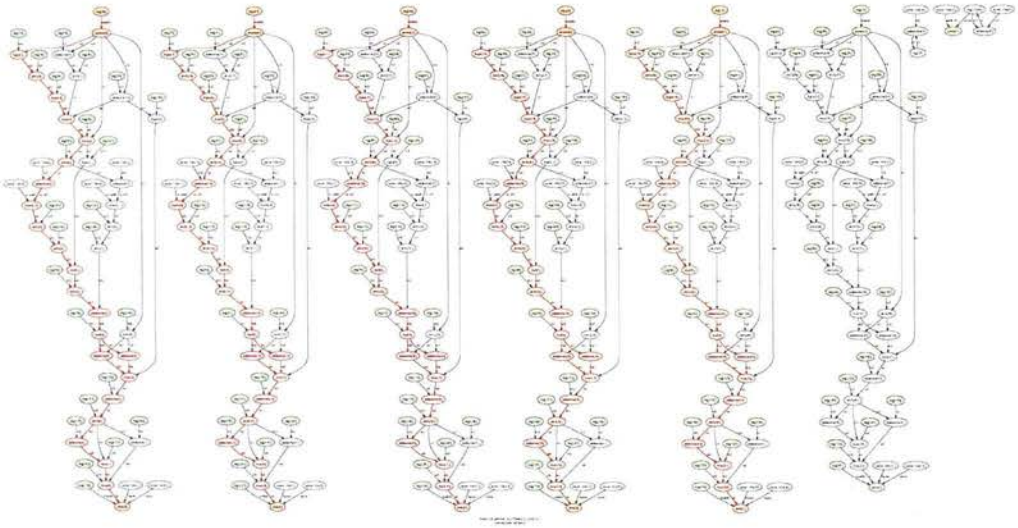


Figure C.5: Data flow graph for the configuration context corresponding to the main loop (kernel) of the gamma correction module used in section 5.8.2 on page 174. Generated by the RICA tools. This version uses combinatorial memory reads (RMEM) for each of the 12 table look-ups. Not pipelined. The critical path is shown by a red outline.

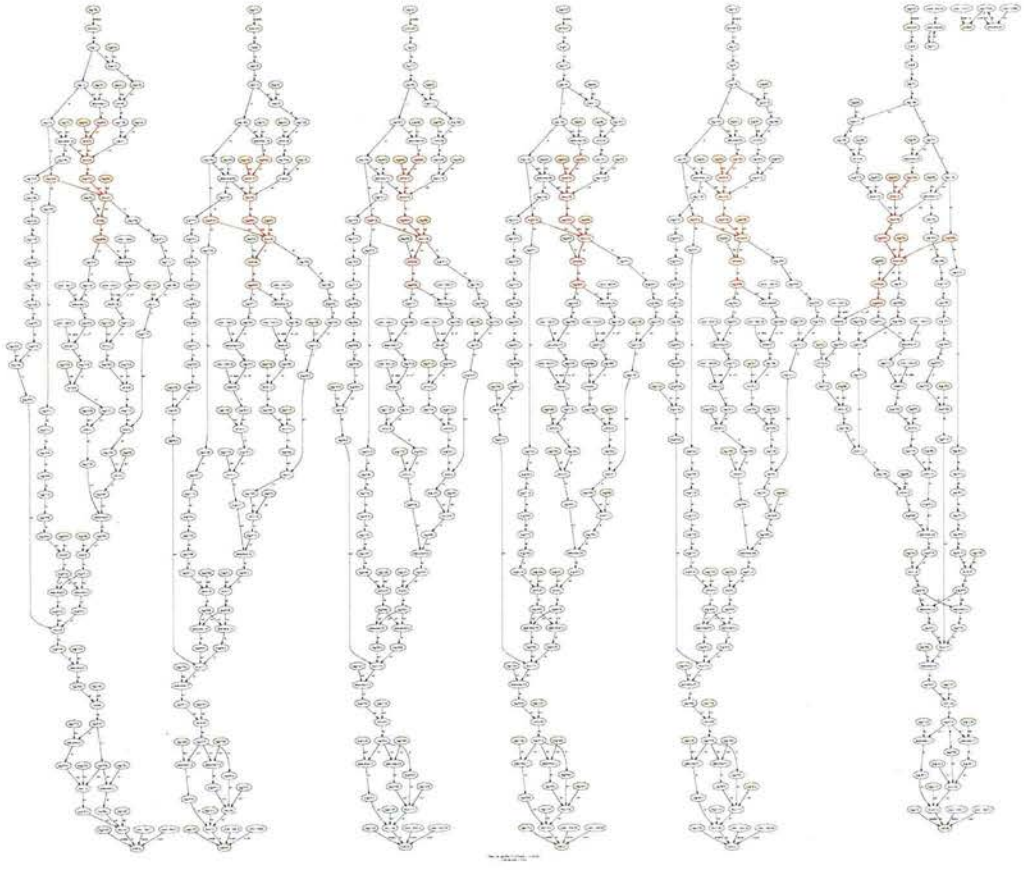


Figure C.6: Data flow graph for the configuration context corresponding to the main loop (kernel) of the gamma correction module used in section 5.8.2 on page 174. Generated by the RICA tools. This version uses combinatorial memory reads (RMEM) for each of the 12 table look-ups. Maximally pipelined (target 6ns). Pipeline registers are shown with a pink fill. The critical paths are shown by a red outline. Note that in this case two adjacent pipeline stages happen to have equal critical paths, so the critical path looks twice as long as it really is.

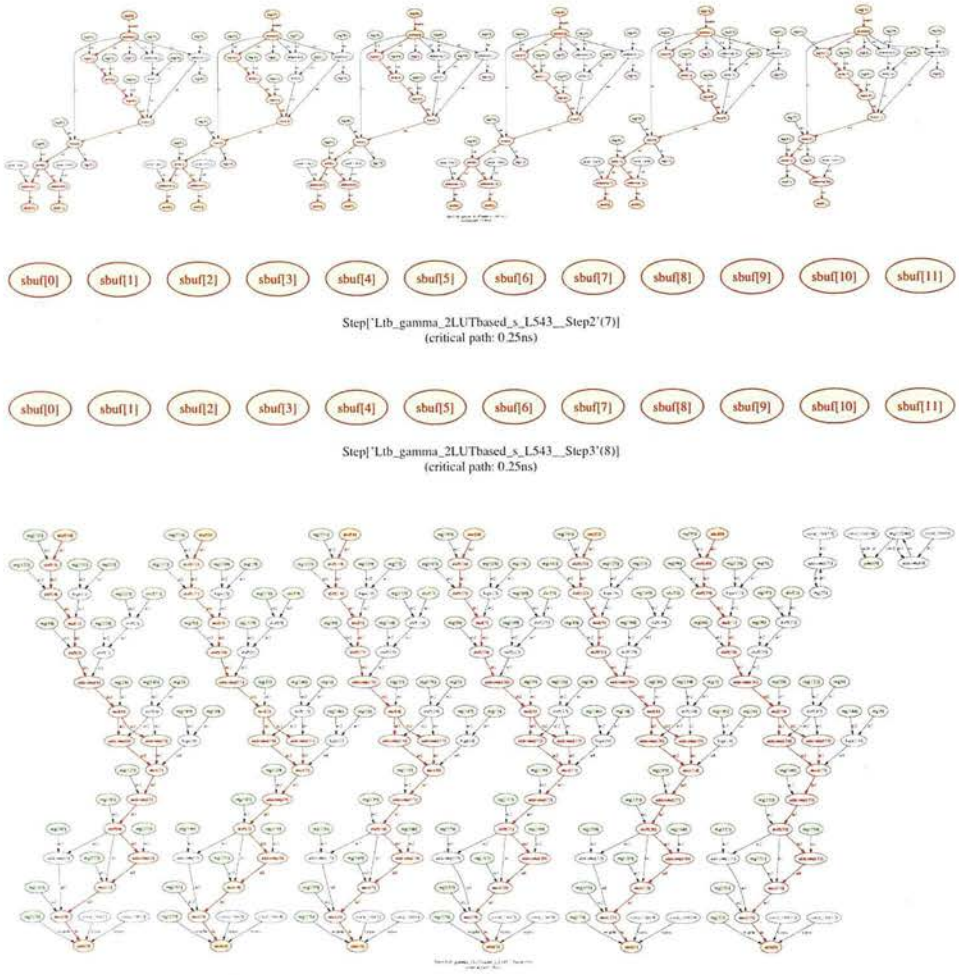


Figure C.7: Data flow graphs for the configuration contexts corresponding to the main loop of the gamma correction module used in section 5.8.2 on page 174. Generated by the RICA tools. This version uses internally pipelined (3 stages) memory reads (SRBUF_RAM) for each of the 12 table look-ups. Not pipelined. The loop has been broken into 4 steps to work around the internal pipelining of the cells, where two of the steps have no connections (just the `sbuf` cells being active, allowing data to propagate internally).

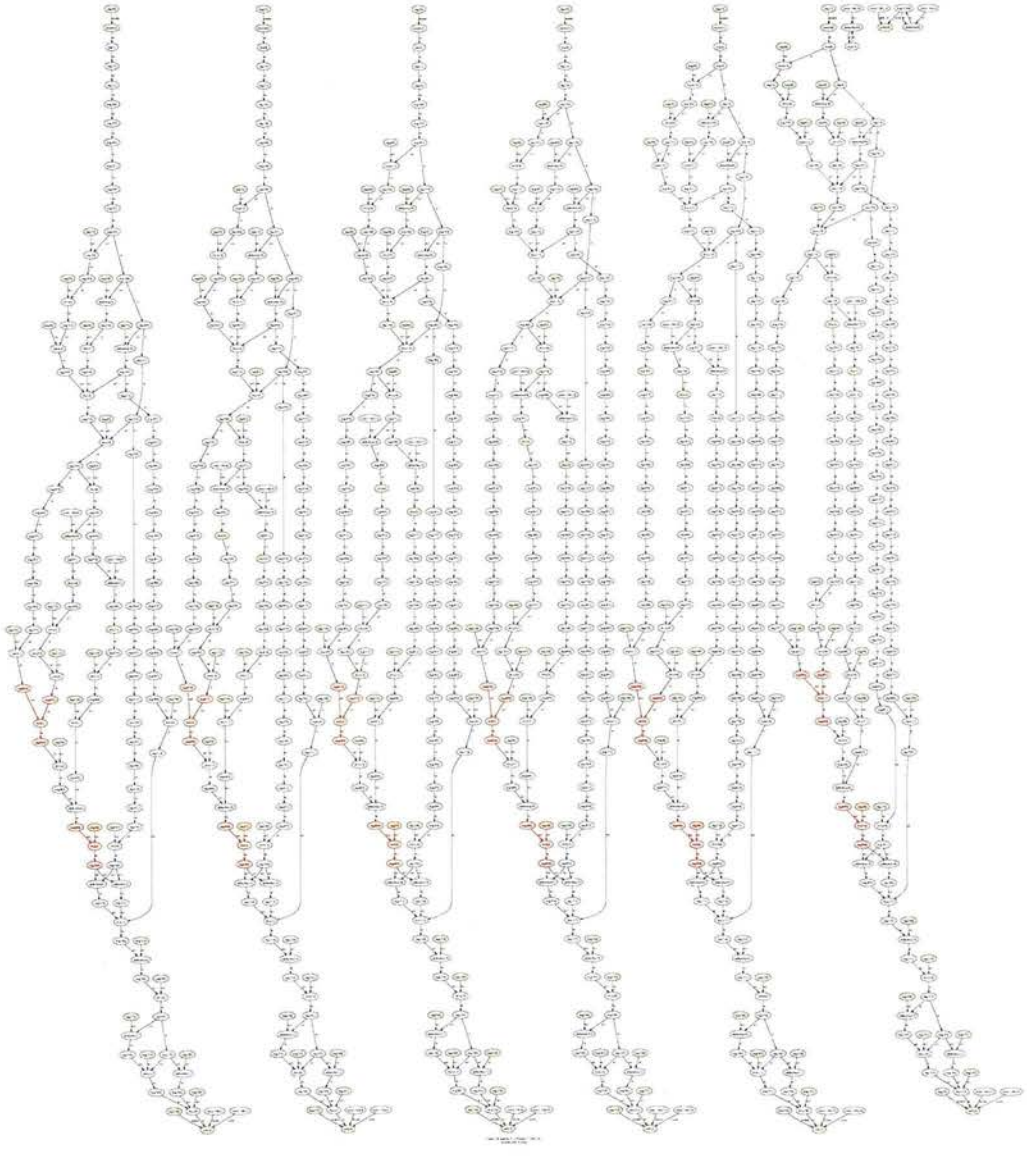


Figure C.8: Data flow graph for the configuration context corresponding to the main loop (kernel) of the gamma correction module used in section 5.8.2 on page 174. Generated by the RICA tools. This version uses internally pipelined (3 stages) memory reads (SRBUF_RAM) for each of the 12 table look-ups. Maximally pipelined (target 5ns). Pipeline registers are shown with a pink fill. The critical path is shown by a red outline.

Appendix D
Publications

Automated Dynamic Throughput-constrained Structural-level Pipelining in Streaming Applications

Mark Muir [1]
[1] The Universtiy of Edinburgh
Mayfield Road, Edinburgh, EH9 3JL
United Kingdom
Mark.Muir@ed.ac.uk

Tughrul Arslan [1,2]
Iain Lindsay [1]
[2] Institute for System Level Integration
Alba Centre, Livingston, EH54 7EG
United Kingdom

Abstract

Stream processing applications such as image signal processing demand high throughput. However, customers increasingly demand runtime flexibility in their designs, which cannot be provided by custom ASIC solutions. Currently, reconfigurable processors tend to offer insufficient throughput for widespread use in streaming applications. This paper demonstrates how structural-level pipelining techniques can be applied to rapidly dynamically reconfigurable computing architectures, in order to increase throughput. This is done by automatically inserting registers into the data path of performance critical code sections that have already been optimised into a single configuration context. A new algorithm is presented to choose the insertion point of pipeline stage registers in order to meet a specified throughput whilst minimising register resource usage. The paper then demonstrates a new approach where properties of dynamic reconfiguration can be utilised to perform the tasks of pipeline stage initialisation and flushing. The technique is demonstrated on a real-life application: the demosaic filter in a standard image signal processing pipe used in modern digital cameras, and can be seen to boost the throughput from 16MPixels/s to 51MPixels/s on an example reconfigurable processor.

1. Introduction

The choice of platform for many modern digital signal processing tasks in embedded systems is often limited to application-specific integrated circuits (ASICs), since off-the-shelf programmable architectures such as DSPs and microprocessors cannot meet the throughput requirements, whereas reconfigurable hardware such as field-programmable gate arrays (FPGAs) require too much area and power. However, for applications that demand an element of reprogrammability, streaming processors (such as those offered by Ambric [1] and SPI [2]) are becoming an increasingly attractive solution, which improve on throughput by providing multiple processing elements/cores with an interconnect structure suited to streaming. However, these processing elements—usually based on regular DSP designs—often equate to significant silicon area. Coarse-grained dynamicaly reconfigurable architectures (DRAs) offer a high degree of parallelism, sufficient to achieve high throughput [3][4]. Thus fewer cores are required for a given application, leading to a much lower area overhead. These coarse-grained architectures are reconfigured very rapidly (e.g. millions of times per second), in order to achieve control flow similar to a regular microprocessor. This paper focuses on maximising the performance of programs running on a single core. However, the techniques can be directly applied to programs running on additional cores in a complete streaming application.

Coarse-grained DRAs, such as instruction cell based computing architectures [5][6], provide a high degree of instruction chaining inside the core, by allowing arbitrary connections to be made between the various functional units via a configurable routing network. This allows quite complex data paths to be rendered onto the fabric and executed in a single configuration. This makes these architectures particularly suitable to stream processing, as fewer fetches from program memory are required. Performance is optimised by attempting to match the size of each kernel (inner loops where most of the execution time is spent) to the available resources, allowing them to fit into a single configuration context. This allows the configuration to persist for many clock cycles, operating on new data on each cycle. This increases throughput, since no time is spent having to reconfigure the core between successive iterations. It also decreases power consumption, as the configuration only needs to be fetched from program memory (or cache) once—upon first entering the kernel—rather than on every iteration. However, the resulting data paths can often have a long critical path, leading to poor temporal utilisation of the functional units, since they have to wait until all functional units have completed before operating on the next batch of data, which limits the throughput.

Pipelining provides a way of starting to operate on a new

batch of data before an old one has completed, so that the functional units of multiple stages of the kernel can be active concurrently; each operating on a different batch of data. This paper describes how structural-level pipelining can be applied *dynamically* to architectures that support a high degree of operation chaining. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at run-time. This has the same effect as adding pipelining in hardware, but can be changed at run-time. Furthermore, these custom pipelines can be initialised and flushed in separate configuration contexts, reducing the resource requirement of the pipelined kernel.

Section 2 reviews existing pipelining techniques, and relevant software optimisation techniques. Section 3.1 describes an algorithm to perform pipeline stage allocation, and section 3.2 shows how properties of dynamic reconfiguration can be used to fill and flush the resulting pipeline. Section 4 shows the result of applying this technique to a real-life kernel used in image processing.

2. Previous work

For architectures that support instruction chaining, scheduling involves mapping as many dependent and independent data paths into as few configuration contexts as possible [7]. Independent data paths run in parallel, so the time for which a configuration persists is determined by the maximum critical path length of these data paths. If sufficient functional unit resources are available, loops can be optimised by loop unrolling [8]—i.e. placing multiple iterations as independent data paths in the same configuration. This allows multiple iterations to begin and end at once. This does not change the original critical path length, yet can increase the throughput. The throughput is determined by the critical path length of a loop iteration and the number of iterations that can be performed at once. During each execution of the loop configuration context, data propagates through the operation chains until the final result is ready. This means that the functional units involved in that chain are only performing useful work for a fraction of the time. This is where structural-level pipelining of these data paths comes in—to artificially reduce the critical path length by allowing new iterations to begin without waiting for the completion of previous iterations.

Various approaches of pipelining data paths have been proposed [9]. These require that the designer specifies a throughput constraint, in order to allow the algorithm to best make the choice between throughput and the area overhead each pipeline stage introduces. These approaches describe various algorithms for the task of pipeline stage allocation, applied to a number of different levels in a design. On reconfigurable architectures such as FPGAs, custom pipelines can be rendered as part of the configuration, leading to significant increases in throughput [10].

Performing this pipelining dynamically as part of the configuration allows the throughput of a given DRA core to be increased, without reducing its flexibility. A generic stream processing engine built from these cores would therefore be able to achieve much higher throughputs over a wide range of streaming applications. For a given throughput requirement, fewer cores are required with this approach, which reduces the area and also the complexity of application development.

3. Dynamic pipelining

Conventional structural-level pipelining can be applied to single configuration context kernels with long critical data paths, in order to reduce the critical path, and thus increase throughput. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at runtime. This is done using existing register resources in the core to delay values for a single execution cycle, allowing values to be bridged across pipeline stage boundaries.

Structural pipelining is applied to the kernel basic block by first assigning each operation in the original data flow graph to a pipeline stage. Then, registers are introduced to store values over boundaries between pipeline stages. Figure 1 shows an example kernel before and after structural-level pipelining.

3.1. Pipeline stage allocation

First, constraints are defined between operations, where the order of execution is important. Examples include 'same stage or earlier' constraints between operations reading from input registers and operations that have those same registers marked as global output registers, and 'same stage or earlier' constraints between data memory read operations and potentially aliasing data memory write operations. All operations in a feedback chain must be placed in the same pipeline stage, since such chains require single-step total latency in order to keep the pipeline full.

The algorithm is a form of list scheduling. Only operations whose predecessors (in the data path) have already been assigned a pipeline stage may be considered for insertion on each pass. In order to minimise the register count, operations should be placed in as late a pipeline stage as possible. Operations that must be placed in the same stage are dealt with together. Operations are considered for placement in the latest pipeline stage containing any of their predecessors. Then, the insertion point is moved towards later pipeline stages until all constraints have been satisfied. Once a valid insertion point has been identified, the critical path is calculated for the resulting (incomplete) configuration context with the operation in that pipeline stage. If the critical path meets the target value, the operation is placed

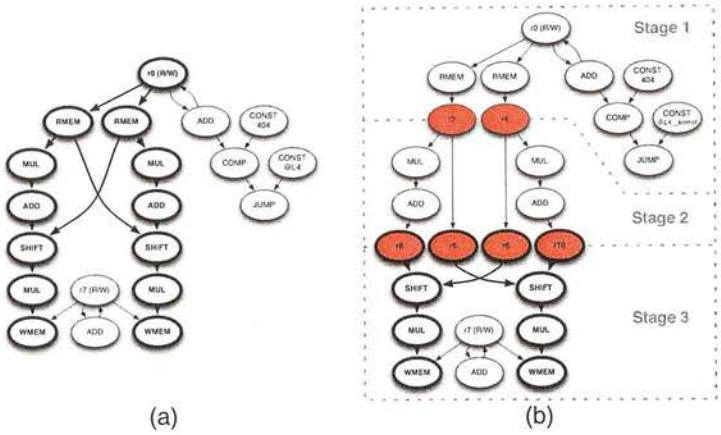


Figure 1: Example kernel data flow graph, (a) before pipelining, (b) after pipelining (kernel loop context). The inserted pipeline stage registers are shown in red. The per-cycle critical path is shown in bold, and is shorter in (b), which allows for a higher throughput.

in that pipeline stage. Otherwise, the operation is added to the next pipeline stage (creating it if it does not exist).

Once the pipeline stages have been determined, pipeline stage registers are assigned as follows: for each pipeline stage in sequence, assign a new register storing the value produced by each operation in all previous pipeline stages that need to be stored for use in this or any later stage.

3.2. Dynamic initialisation and clean-up

Normally, a pipelined design would require additional logic to take care of initialising the pipeline stages, or to suppress the operations in later pipeline stages until the previous stages have filled (predication), so that they do not operate on garbage. However, since the pipelines in a coarse-grained DRA are themselves rendered as part of the configuration context. Provided that the configuration time is not significantly larger than the execution time of each step, dynamic reconfiguration can be used to render different configurations before the main kernel loop configuration, to fill successive stages of the pipeline, and similarly to flush the pipeline after exiting the kernel loop. This allows the kernel loop configuration to assume that the pipeline stages are always full.

Prologue: New configuration contexts are created to initially fill each successive stage of the pipeline. For n pipeline stages, $n-1$ pipeline filling contexts are created.

Loop: A single configuration context is created for the kernel loop, which includes all pipeline stages.

Epilogue: New configuration contexts are created to flush successive stages of the pipeline. For n pipeline stages, $n-1$ pipeline flushing contexts are created.

The core is dynamically reconfigured to first perform pipeline initialisation, then reconfigured to execute the kernel loop, then finally reconfigured to flush the pipeline—as demonstrated in figure 2. This is similar to the epilogue and prologue in software pipelining [11].

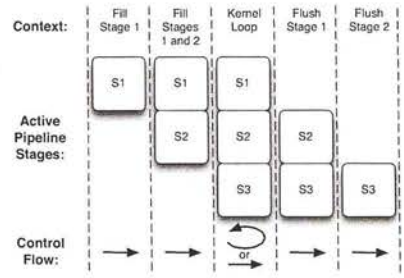


Figure 2: Control flow for a 3-stage pipelined kernel, showing which stages are active in each context (and moment in time). Execution flows from one context to the next, except in the kernel loop, which loops back to itself (holding the same context) until the end condition is satisfied.

Figure 2 shows which stages of the pipeline are active during execution for a 3-stage pipeline. As the target architectures may not be state free (e.g. memory access), it is important to not allow any operation in any pipeline stage to operate on garbage, and to preserve the execution count. With the arrangement shown in the figure, all pipeline stages will be executed the same number of times irrespective of the number of iterations performed in the kernel loop.

4. Application to streaming

The algorithm described in this paper was applied to a real-life application: a 3-line demosaic filter [12], which in-

volves interpolating missing colour components from the Bayer output of a colour filter array sensor. This is a high-throughput task normally done on-chip (integrated into the sensor) as part of a custom image signal processing pipeline, used in modern digital cameras and mobile phones. This is typically the most computationally intensive part of a standard Image Signal Processor (ISP). The filter was re-implemented on a reconfigurable instruction cell-based processor [5], using the C language. Software optimisation techniques were used to reduce the filter kernel into a single basic block, small enough to fit onto the target architecture in a single configuration context. The throughput of the resulting filter is given in table 1.

The operations of the resulting kernel were then pipelined using the algorithms described in this paper, for several target critical path lengths (timing constraints), the results of which are also given in table 1.

Target critical path (ns)	None	40.0	30.0	20.0	19.0	16.0
Actual critical path (ns)	60.3	38.4	29.2	21.3	20.3	19.6
Throughput (MPixels/s)	16.4	26.0	34.2	47.0	49.3	51.2
Pipeline stages	-	2	3	4	6	6
Additional registers	0	10	17	27	31	34
Additional contexts	0	2	4	6	10	10

Table 1: Performance of the demosaic filter kernel before pipelining, and after pipelining. Throughput and additional register and program memory resource requirements are shown.

Pipelining can be seen to increase the throughput, at the expense of extra registers, and additional program memory for the prologue and epilogue. The last column in table 1 shows that a natural throughput limit is reached, determined in this case by the length of the feedback chains present in the kernel data flow graph. Note that for a target of 19.0ns in this case, the resulting critical path is greater than that obtained for a target of 16.0ns. This represents boundary noise in the register stage allocation algorithm, where depending on previous choices, a different local minimum may be found.

5. Conclusions

This paper demonstrates that structural-level pipelining techniques can be applied via software to rapidly reconfigurable/programmable architectures supporting operation-chaining, where complete kernels can be mapped into a single configuration context/cycle. This improves throughput by reducing the critical path length of the looping kernel. This makes such architectures ideal candidates for use as the cores in a stream processing engine, as fewer cores are needed to meet a particular throughput. This work concentrated on reconfigurable instruction cell processors, which support a high degree of operation chaining. However, the same techniques could be applied to other quite different architectures that support operation chaining, such as upcoming VLIW/ULIW processors.

Furthermore, this paper introduced the idea of achieving pipeline filling and flushing through dynamic reconfiguration, in a manner similar to that used in software pipelining. Pipelining was shown to increase the register requirement, and uses more program memory to store the additional configuration contexts (prologue and epilogue). However, the program memory overhead can be largely avoidable, since the additional contexts are suited to temporal compression. The potential throughput is limited by the number of registers available for use in connecting the pipeline stages, and by the presence of feedback loops that demand single cycle latency (e.g. when updating the value of a register). The algorithm was applied to a demosaic filter for a variety of target throughput constraints, and achieved a maximum throughput of more than three times that of the original kernel.

References

[1] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, 2007, pp. 55–64.

[2] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Daly, "A programmable 512 GOPS stream processor for signal, image, and video processing," in *Solid-State Circuits Conference*, 2007, pp. 272–602.

[3] A. Major, T. Arlsan, *et al.*, "H.264 decoder implementation on a dynamically reconfigurable instruction cell based architecture," in *International SOC Conference*, 2006, pp. 49–52.

[4] Z. Khan, T. Arlsan, *et al.*, "Implementation of a real time programmable encoder for low density parity check code on a reconfigurable instruction cell architecture," in *Design Automation Conference, Asia and South Pacific*, 2007, pp. 583–588.

[5] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 1–11, 2008.

[6] "Loosely-biased heterogeneous reconfigurable arrays," U.S. Patent 20 050 257 024, 2005.

[7] Y. Yi and I. Nousias, "System-level scheduling on instruction cell based reconfigurable systems," in *Design Automation and Test in Europe, International Conference on*, 2006, pp. 381–386.

[8] J. Sanchez and A. Gonzalez, "The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures," in *ICCP Parallel Processing, International Conference on*, 2000, p. 555.

[9] S. Bakshi and D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 4, pp. 419–432, 1999.

[10] S. Silva and S. Bampi, "Area and throughput trade-offs in the design of pipelined discrete wavelet transform architectures," in *Design Automation and Test in Europe, International Conference on*, 2005, pp. 32–37.

[11] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *ACM SIGPLAN conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.

[12] J. Mukherjee, M. Moore, and S. Mitra, "Color demosaicing with constrained buffering," in *Signal Processing and its Applications, Sixth International Symposium on*, vol. 1, 2001, pp. 52–55.

EXTENSIBLE SOFTWARE EMULATOR FOR RECONFIGURABLE INSTRUCTION CELL BASED PROCESSORS

Mark Muir[1,3], Iain Lindsay[1], Tughrul Arslan[1,2,3], Ioannis Nousias[1,3], Sami Khawam[3], Mark Milward[3], Nazish Aslam[2,3], Adam Major[1,3]

[1] The University of Edinburgh
[2] Institute of System Level Integration
[3] Spiral Gateway
contact: mark.muir@ed.ac.uk

ABSTRACT

This paper presents a novel high-speed behavioural simulator (software-based emulator) for reconfigurable instruction cell based processors. These architectures are particularly suited to providing low-power, low-cost implementations of applications in a streaming environment, such as image signal processing, video playback, or base-band signal processing. As a result, many realistic applications operate on very large data sets, so simulation time plays a key role in the time to market. The key aspect of this work is an efficient serialisation algorithm (based on topological sort), able to capture the intricacies of reconfigurable processors that can be reconfigured very rapidly (ns). This allows for a new generation of high-speed emulation models to be constructed. The performance of this algorithm deployed in an interpreter-based model is compared to other simulation techniques. The emulator can achieve performance around two orders of magnitude higher than current event-driven software models, and similar to that of an FPGA-based model. This brings the simulation times low enough to be able to use this technology as the basis for feedback-directed optimisation, which can significantly improve the performance of application code.

I. INTRODUCTION

Reconfigurable instruction cell based processors [1] are coarse grained reconfigurable computing fabrics, for use in embedded systems. These reconfigurable architectures fill the gap between traditional field programmable fabrics (such as FPGAs) and microprocessors. These architectures (introduced in section III) are an emerging technology, and their designs are still being actively explored. Simulation is needed to allow for rapid modification and evaluation of the core design, avoiding the time needed to re-implement and test the core using a hardware description language (HDL) for an FPGA implementation, or the cost of re-fabricating the array. Furthermore, these architectures are intended to be provided as flexible IP blocks, where the end-user can make significant changes to the make-up and functionality of the core. The end-user expects a complete toolchain to be available that is able to reflect these changes, in order for the complete hardware/software design space to be explored. Such a toolchain normally consists of an optimising compiler, and a simulator [2, 3]. The application domains that these architectures are mainly aimed at tend to operate on large data sets, such as video playback (H.264 decoding [4]), digital signal base-band processing [5], and image signal processing [6]. As a result, simulation time is a crucial factor in determining the length of the architecture definition cycle, and thus time to market.

The similarities to a microprocessor mean that software-based simulation technologies traditionally used with microprocessors can be adapted for these new architectures, by taking account of the parallelism in the array. Section II reviews traditional microprocessor emulators, and their uses. However, reconfigurable architectures support operation chaining—the ability to execute dependent and independent instructions within the same clock cycle / configuration context—which traditional emulation technology cannot model.

Modelling parallelism on a serial machine has already been addressed in HDL simulation, particularly those intended for dynamic reconfiguration [7]. These concepts are borrowed to derive an event-driven model that captures the data paths between processing elements in the array. SystemC provides an object-orientated event-driven model with a kernel similar to an HDL simulator, but described only at the behavioural level in C.

This kernel-based approach of serialising in response to run-time events imposes an overhead per configuration context. For traditional reconfigurable and dynamically reconfigurable hardware, the rate of reconfiguration is low, so the overhead of updating the event-driven model on each configuration context represents only a small fraction of the total execution time. However, reconfigurable instruction cell based processors are reconfigured many *millions of times per second*, so this overhead introduced by the model is large compared to the actual work done by the operations of the modelled cells.

Therefore, moving this overhead into a pass prior to program execution is highly desirable. This is what the software-based emulator presented in this paper does. The emulator moves away from the event-driven approach, and instead mimics the same order of data flow by generating a static schedule of operations that are performed sequentially. The algorithm for generating this schedule, along with the required storage queues, is described in section VI. This is a new extension to traditional software-based emulator technology, allowing this type of model to work with these emerging architectures. In section VII, the run-time per-

formance of the proposed software-based emulator is compared with that of a SystemC event-driven simulator, and an FPGA implementation of the instruction cell array. A set of representative applications are run on all three models. Section VIII concludes.

II. EMULATION

Software-based emulation of microprocessors has been used since at least the 1970s [8]. Emulation models the instruction set of the target architecture by mimicking the way that the state of the CPU, registers, and memory is affected by each operation in the instruction set. The fetch and execution of instructions in the emulator is performed in the same sequential manner as in the target CPU. Traditionally, such emulators have been custom-built to a particular target architecture and platform [9]. Since most CPUs are conceptually similar, these concepts can be abstracted, making the emulator extensible. This is commonly achieved through object-orientated design [10, 11]. Emulators are part of many modern commercial tool sets [12]. Emulation sees the following uses:

Behavioural validation: the target architecture and associated application development toolchain can be proven before committing to silicon, or dedicating time to detailed HDL simulation.

Product/Application demonstration: the ability to add emulated hardware allows for applications to be demonstrated in near real-time, before the hardware is available.

Provides an easily modifiable test bench: adding emulated hardware at the behavioural level aids in developing peripherals, since these can be added to the emulator, and their usefulness or interface design explored. This makes it is easy to try out new ideas (platform exploration), without having to design them beyond the behavioural level.

Reduces development time: algorithms can be tested and timing information estimated in a fraction of the time of other software-based simulation techniques available.

Feedback-directed optimisation: information can be extracted about a program through profiling during execution on the emulator. This information can then be used by a compiler [13] to make more informed decisions when applying optimisation [14].

The generalisation of traditional emulation concepts has also extended to the point where emulators can be automatically generated from an abstract machine description, along with an optimising compiler/scheduler as part of a retargetable toolchain [2]. Machine description languages have progressed to the extent that features of increasingly complex architectures can be captured, including deep pipelining of functional units, multiple instruction issue, and the design of the memory subsystem [3]. However, these languages are not yet able to capture the operation chaining available in reconfigurable processors, except by enumerating every possible configuration, which would be impractical. However, such languages could be extended to capture this information, and such a description could be used to automatically generate a simulator using the technology presented in this paper.

Developments in modern compiler technology have exhausted much of the potential for static optimisation, and so the trend is a shift towards feedback-directed optimisation. As a result, an emulator for this purpose is likely to become a significant part of standard toolchains. With this in mind, the speed of simulation directly affects the scalability of the toolchain with respect to target applications, which are of ever increasing complexity. Hardware acceleration has been commonly explored for use with emulation [15, 16, 17]. However, several of the uses listed above make the requirement of additional hardware undesirable (if not impractical), and so a software-only solution is the main focus of this paper.

III. RECONFIGURABLE INSTRUCTION CELL BASED PROCESSORS

Reconfigurable instruction cell based processors [1] are coarse grained reconfigurable computing fabrics, consisting of a heterogeneous array of programmable cells on a programmable interconnect network. An example can be seen in fig. 1. The cells perform operations similar to those found in a conventional arithmetic logic unit (ALU), and can be combined through the reconfigurable interconnect to perform more complex instructions in a single configuration cycle (context). A configuration context persists for a time sufficient for the sequence of connected cells with the longest propagation delay (the critical path) to complete, and then the next configuration context is loaded. The next configuration context can be chosen arbitrarily, by programming the jump cell. This way, arbitrary program control flow is possible. Essentially, the architecture can look like a multiple-issue microprocessor, with a very large instruction set. A reconfigurable fabric exploits parallelism and thus can achieve higher performance than other reprogrammable technologies, like microprocessors or DSPs. It essentially offers the programmability of a microprocessor-based solution, but with power consumption and performance approaching that of an ASIC.

IV. THE MODELLED SYSTEM

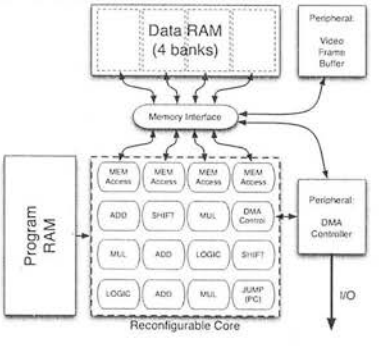


Figure 1: Modelled system: reconfigurable core (simplified), memory, and example peripherals.

An example system that can be modelled with the emulator is shown in fig. 1, and consists of the instruction cell array core, with separate program and data memories, and some simple peripherals. In this example, data memory is arranged in multiple banks, accessed through special cells in the array. Since more than one memory access cell is provided in the array, multiple accesses can be performed by the core in one configuration context. If all such accesses are to different banks, then these accesses are performed in parallel. Otherwise, conflicting requests are performed sequentially, which incurs a dynamic delay. The emulator can be used to characterise memory access patterns and use the results to direct scheduling [18] and linking of the program to optimise access to data memory (feedback-derived optimisation, mentioned in section II).

V. EXTENSIBILITY

Software emulations of memory-mapped peripherals such as a DMA controller, video frame buffer, or audio buffer can easily be added. These communicate with the core just like they would in real life: either through the memory interface, or through special-purpose cells in the array. New instruction cells can be added to the core simply by defining a new object. New memory-mapped peripheral modules can be added by defining a new object for the peripheral, which responds to events from the memory interface through known method calls, see fig. 2, in response to activity on the appropriate addresses. Peripherals can have a kernel that operates on a separate thread, if they are to perform operations that are independent to the core. The video frame buffer emulation is an example of this: it performs colour space conversions and renders frames largely on its own time-base. More complex peripherals, such as a DMA controller, can be added that connect to both the memory interface and to the array via special control cells. These could be implemented by creating a new object for the special cell, and allowing the cell object to communicate with the memory interface. Other scenarios are also possible.

```
object Memory_interface
{
  properties:
  - references to the data memory banks

  methods:
  - readFromAddress:
  - writeData:ToAddress:
}
```

Figure 2: Pseudo-code for memory interface.

VI. EMULATOR TECHNOLOGY

The emulator is an object-orientated program written in C++, and is modular in design. Each hardware component mentioned in section IV is represented by a class (object), and they communicate with each other via method calls. The model of the core is simply a set of instruction cell models, each of which contains the state information that the real cell would maintain, and a set of cell 'actions' which capture the behaviour of that cell. The cell actions are implemented as C++ 'methods'. The operation of a given cell is represented by one or more of the following cell actions:

Evaluate: Assign the output value of the cell and/or modify the internal state of the cell according to the configuration word.

Operate: Assign the output value of the cell according to the configuration word and the values read from its input(s).

Update: Modify the internal state of the cell according to the configuration word and values read from the input(s).

A serialised configuration context consists of the 'evaluate' actions (scheduled in any order), followed by the 'operate' actions (specifically ordered by the serialisation algorithm described in section A.), followed by the 'update' actions (in any order). Cells that perform only simple combinatorial operations—which calculate an output value based on the values of their inputs—implement only the 'operate' action. The code sample in fig. 3 demonstrates a simplified version of an 'ADD' cell, which is an example of a combinatorial cell.

```
object Add_cell extends Instruction_cell
{
  properties:
  - output // Storage for cell's output.

  constructor:
  - define cell configuration and input ports.

  methods:
  - operateWithConfiguration:inputs:
  {
    switch configuration
    {
      case ADD_ADD_SI: // Single integer.
        output = in1 + in2
        case ADD_SUB_V2HI: // Vector mode.
          output = (in1[1] - in2[1],
                    in1[0] - in2[0])
      etc.
    }
  }
}
```

Figure 3: Simplified ADD cell class implementation pseudo-code.

The emulator parses the netlist describing the target program, then serialises the operations of each configuration context into a sequence of equivalent cell actions. These serialised operations are stored in an internal data model. The serialisation process is described in section A.. Execution of the program then proceeds: these sequences of cell actions for each configuration context encountered are executed in a large state machine by calling the appropriate virtual function, as shown in fig. 4.

The model of each cell contains a variable that holds the value for the cell's output port. This can then be referenced (read) by the actions of cells that depend on that value (the 'input' vector passed into the operate() method). Note that the program counter can also be updated via the cell actions (for the jump cell), and this determines which configuration context will follow. A configuration context is the smallest

```
// Execute steps until end condition is detected.
do
{
  step index = jumpcell program counter value
  this step = program[step index]
  for each cell action in this step
  {
    switch cell action type
    {
      case Evaluate:
        action.instruction_cell->
                  evaluate(action.configuration)
      case Operate:
        action.instruction_cell->
                  operate(action.configuration,
                          action.inputs)
      case Update:
        action.instruction_cell->
                  update(action.configuration)
    }
  }
}
while jump cell hasn't detected end
```

Figure 4: Core execution loop pseudo-code.

unit that can be used as the target for jumps. The data memory is modelled as a simple array wrapped by an object that provides an interface to read and write to the memory, as described in section V.

A. *SERIALISATION ALGORITHM*

The serialisation algorithm is used to create the internal representation which drives the execution state machine. The key requirement of this algorithm is to ensure that the result of executing the sequence of cell actions in the execution model, exactly matches the result of the original data flow graph for that configuration context. This simply requires that a cell's action (for calculating its output value) is scheduled before those of any dependent cells (successors). The serialisation algorithm requires extension to deal with situations where cells maintain internal state from one configuration context to the next. To explain this, we first give an example involving only combinatorial cells, then a second example showing the extension required to avoid apparent connection loops arising from internal state.

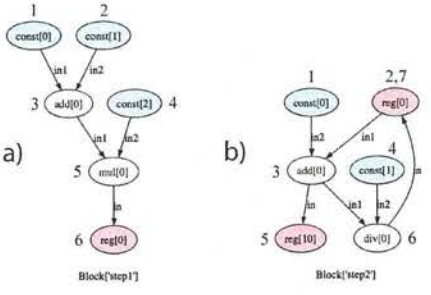


Figure 5: Example configuration contexts: (a) involving only combinatorial operations, (b) including a 'connection loop'—this case is valid since the loop involves a register, which is a 'terminal cell'.

The data flow graph example for a configuration context involving only combinatorial cells is given in fig. 5(a). The operation of any purely combinatorial cell needs only an 'operate' action to be defined. The constant cells supply the operands for a set of operations, and the final result is written to storage. A human might choose a sequence such as that shown by the numbers in fig. 5(a). The algorithm employed by the emulator constructs the connection hierarchy between the active cells as a directed graph. Once the hierarchy is complete, the topological sort operation from graph theory is applied to the graph. The topological sort results in nodes being ordered in descending order of depth in the connection hierarchy. The ordered result is used to schedule the 'operate' actions of each cell. Within a given depth, the cell actions could be scheduled in any order, without affecting the overall result. The direction of the arrows in fig. 5 indicates the direction of data flow, and defines the terminology of predecessor feeding data to a successor, i.e. one of the successor's input ports is connected to the output port of the predecessor. In the completed hierarchy, a predecessor lies in some level lower than that of any of its successors.

Things are a bit more complicated than this, however, because some cells maintain internal state information. Taking registers as an example, the output of the cell does not depend on the input in the current configuration context; instead it depends on the internal state of the register cell (which in turn usually depends on the input to the register from a previous configuration context). This means that it is valid for a register to appear in a 'connection loop'—where the output of the register is used in some sequence of operations, the result of which is stored back in the same register. This results in a cyclic graph, making a topological sort impossible. Essentially, the register cell can be thought of as two cells—one emitting the current value, and one receiving the new value. However, this is not a clean approach.

Alternatively, we can introduce the concept of 'terminal cells'—i.e. cells where the inputs do not affect the outputs during the execution of that configuration context. Now, connection loops are valid if one of the cells in the loop is terminal. Terminal cells provide an 'evaluate' action, in addition to an 'operate' action. Calculating the output value of a terminal cell can always be done before anything else during the execution of a configuration context (since the value does not depend on the result of any other cell during that configuration context), and writing to the input(s) of a terminal cell can always be done after anything else during the execution of a configuration context (since the written value does not affect any other cells during that configuration context). Furthermore, some cells need to have their state modified upon each configuration context transition (reconfiguration). This is done by providing an 'update' action, that is performed once the rest of the actions have been executed. So, the algorithm is extended by scheduling all 'evaluate' actions first, followed by the sequence of 'operate' actions obtained from the topological sort, and finally all 'update' actions are scheduled. fig. 5(b) shows an example, to which the algorithm would assign the following

sequence of cell actions:

```
const[0](evaluate), const[1](evaluate), reg[0](evaluate),
add[0](oper.), div[0](oper.), reg[0](oper.), reg[10](oper.),
reg[0](update), reg[10](update)
```

Registers are only a simple example of this problem. More complex examples include interfaces to streaming memories, and cells that are internally pipelined such that their output is delayed by (several) iterations. It has so far proven possible to map all supported cells to this mechanism, and this approach is quite effective in minimising the number of operations that need to be performed for each configuration context.

VII. PERFORMANCE

The performance of the emulator was compared against a SystemC transaction-level model of the same instruction cell-based processor, and an FPGA implementation of the same array (i.e. a dynamic reconfigurable fabric on a static reconfigurable fabric). A quad 2.2GHz AMD Opteron PC was used as the host machine for the emulator and SystemC model. The FPGA used was the Virtex-4 LX 160. Note that an FPGA implementation performs the same role as an HDL simulation of the processor architecture, and is used instead of an HDL simulation since it achieves much higher run-time performance, and so is much more suitable for the task of near real-time application demonstration. The execution speed was used as the measure of performance. The reconfigurable array is intended to have a system clock of 500MHz. The maximum achievable clock on the FPGA implementation of the target processor is 12MHz (determined by the critical path of the synthesised instruction cell array rendered on the FPGA, *which is the same irrespective of the target application*). The ratio of these gives the performance value for the FPGA. For the other methods, the execution time was accurately measured and averaged over several runs. The averaging is necessary for user-space programs, in order to reduce random error introduced by preemptive context switches on the host. Execution speed is the time that the target application should have run for on the reconfigurable array, divided by the average run time on the model. The following algorithms/applications were used:

- Discrete Cosine Transform (DCT) (for MPEG4/H.264 video).
- Finite Impulse Response (FIR) digital filter.
- Dhrystone (integer CPU performance bench-mark).
- MP3 (MPEG-1 layer 3) audio decoder (libmad).
- H.264 video decoder (ffmpeg).

Table 1 shows that the performance of the emulator described in this paper is good compared to the other simulation methods described. The real silicon (native) is between 21 and 101 times faster than the emulator, and the FPGA model is close in speed to the emulator. Since the FPGA is a model of the real silicon, it is a constant fraction of the speed of the real silicon. Both software models vary in execution speed (compared to the real silicon), depending on the application.

	Emulator	SystemC model	FPGA model	Native
FIR	1.000	3.40e-3	0.52	21
DCT	1.000	5.52e-3	1.47	61
H.264	1.000	9.44e-3	1.43	59
MP3	1.000	12.00e-3	2.43	101
Dhrystone	1.000	76.00e-3	0.83	34

Table 1: Execution speed for various standard benchmarks, normalised to the speed of the emulator.

The relative performance of the emulator and SystemC model can also be seen to depend on the application. Since these two models use very similar cell implementations, written in C, this highlights the differences in the overheads incurred by the method of simulation. In addition to performing the actual work of the cells, the SystemC kernel incurs an overhead for each event generated by the active cells, and a further overhead at the end of each configuration context. The emulator on the other hand, only incurs the latter overhead, since everything except for the path of program execution is serialised prior to execution. The Dhrystone example consists of many short basic blocks, which results in very low core utilisation. This represents the extreme of frequent configuration context changes with few cell operations in between. The FIR example represents the opposite extreme, where the program consists largely of one basic block, which results in very high core utilisation, and much core activity between configuration context switches. The results in table 1 show that the emulator is best advantaged when core utilisation is high, which supports this argument.

	Total ops. per iter.	Critical path	Relative speed
Parallel	26	16ns (5 ops.)	1246x
Combinatorial	26	40ns (9 ops.)	1377x
Sequential	11	16ns (5 ops.)	2258x

Table 2: Complexity and relative execution speed (emulator v.s. SystemC model) for some simple test programs written to investigate the reason for the application-dependent relative execution speed.

To examine this further, some small test programs were written, each consisting of a single loop mapping to a single configuration context. In each case, the loop body consists of a relatively simple sequence of arithmetic operations to apply to each member of a data set. The programs differ in when the operations for a given member of the data set are executed. The programs are shown in table 2. To test the effect of the number of events generated per iteration, one program ('Parallel') performs the operations of four members of the data set in parallel; whilst another program ('Combinatorial') also operates on four members of the data set per iteration, but a data dependency exists preventing them from running entirely in parallel (however they still overlap to a certain extent). The number and type of operations performed per iteration in both of these programs is the same; however the latter ('Combinatorial') case has a longer critical path. The relative performance of the emulator and SystemC model is similar for both programs. The execution time of the emulator should depend

only on the operations performed, and not the order. For the SystemC model, the longer critical path (and number of operation chained together) causes more flutter as the combinatorial paths stabilise, resulting in more transition events being generated. However, the execution time for each event is very small compared to the time taken to schedule the events, and the results in fact show a slight relative gain. This indicates that the run-time scheduling is easier when the timing of the events is more sequential. To test the effect of the number of operations per iteration, another program was written ('Sequential'), this time with only one member of the data set operated on per iteration of the kernel. This requires that four times as many iterations are performed. A significant increase in the relative speed of the emulator can be seen compared to the previous test programs. This therefore indicates that the SystemC model incurs a disproportionately large overhead per iteration, which supports the earlier observation with the standard benchmarks.

VIII. CONCLUSIONS

Existing software-based methods of simulation for reconfigurable computing architectures are event-driven, and incur a sizeable time penalty for every configuration context. For instruction cell architectures, which have to be reconfigured many millions of times per second, this overhead eclipses the actual work done by the modelled processing units. A novel approach was suggested to reduce this overhead, by moving the resolution of the dependencies in the data paths into a pre-processing stage, prior to execution. When applied to an example processor, the results (section VII) show that the execution speed achieved using this new approach is around two orders of magnitude higher than an equivalent SystemC model, and largely matches the speed of an FPGA model of the target reconfigurable instruction cell array.

This level of performance makes the proposed emulator suitable for use in feedback-directed optimisation, and thus could be an important part of future toolchains. Furthermore, the emulator is highly adaptable to different types of reconfigurable processors with different functionality but similar control concepts, making it a good candidate for use in retargetable toolchains for hardware/software co-design. In addition, the serialisation algorithm can be applied directly to translation, allowing even faster emulations to be performed. This would be achieved by using the output of the serialisation algorithm to generate static call lists for each configuration context, which are then fed into an optimising linker (such as LLVM [19]) to generate an optimised, native binary, eliminating the overhead of interpretation.

IX. REFERENCES

[1] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 1–11, 2008.

[2] A. Hoffman, T. Kogel, and H. Meyr, "A framework for fast hardware-software co-simulation," in *Design Automation and Test in Europe, international conference on*, 2001, pp. 760–764.

[3] A. Halambi, P. Grun, *et al.*, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Design Automation and Test in Europe, international conference on*, 1999, pp. 485–490.

[4] A. Major, T. Arlsan, *et al.*, "H.264 decoder implementation on a dynamically reconfigurable instruction cell based architecture," in *International SOC Conference*, 2006, pp. 49–52.

[5] Z. Khan, T. Arlsan, *et al.*, "Implementation of a real time programmable encoder for low density parity check code on a reconfigurable instruction cell architecture," in *Design Automation Conference, Asia and South Pacific*, 2007, pp. 583–588.

[6] M. Muir, T. Arslan, and I. Lindsay, "Automated dynamic throughput-constrained structural-level pipelining in streaming applications," in *Design Automation and Test in Europe, international conference on*, 2008, p. TBA.

[7] P. Bellows and B. Hutchings, "JHDL - an HDL for reconfigurable systems," in *FPGAs for Custom Computing Machines, IEEE symposium on*, 1998, pp. 175–184.

[8] L. Robertson, "Anecdotes," *Annals of the History of Computing, IEEE*, vol. 27, no. 2, pp. 82–84, 2005.

[9] H. Diab and I. Demashkieh, "A reconfigurable microprocessor teaching tool," in *Science, Measurement and Technology, IEE Proceedings*, 1990, pp. 287–292.

[10] C. Cooper and P. Werstein, "The use of Java to develop a microprocessor emulator," in *Software Engineering: Education and Practice*, 1998, pp. 272–277.

[11] W. Zaatar and G. E. Nasr, "An implementation scheme for a microprocessor emulator," in *ICECS Electronics, Circuits and Systems, 7th international conference on*, 2000, pp. 169–172.

[12] S. Bush, "ARM offers real-time prototyping capability," *Electronics Weekly*, no. 39339, July 2006.

[13] E. R. Altman, S. Sathaye, and M. Gschwind, "Execution-based scheduling for VLIW architectures," in *Euro-Par'99—Parallel Processing, international conference on*, 1999, pp. 1269–1275.

[14] R. Cohn and P. G. Lowney, "Feedback directed optimisation in compaq's compilation tools for Alpha," in *Proceedings of the 2nd ACM Workshop on Feedback-directed optimisation*, 1999.

[15] M. Gschwind, V. Salapura, and D. Maurer, "FPGA prototyping of a RISC processor core for embedded applications," *IEEE Transactions on VLSI Systems*, pp. 241–250, 2001.

[16] Y. Nakamura and K. Hosokawa, "Fast FPGA emulation based simulation environment for custom processors," *IEICE transactions on fundamentals of electronic communications in computer science*, vol. E89-A, pp. 3464–3470, 2006.

[17] S. Fink and E. Sanchez, "Development and prototyping for an 8-bit multitask micropower processor," in *Proceedings of the 6th IEEE International Workshop on Rapid System Prototyping*, 1995, pp. 75–78.

[18] Y. Yi and I. Nousias, "System-level scheduling on instruction cell based reconfigurable systems," in *Design Automation and Test in Europe, international conference on*, 2006, pp. 381–386.

[19] C. Lattner and V. Adve, "The LLVM compiler framework and infrastructure tutorial," in *Mini Workshop on Compiler Research Infrastructures (LCPC'04)*, 2004.

Automatic Dynamic Structural-level Pipelining in Reconfigurable Processors

Mark Muir[1], Nazish Aslam[2],
Ioannis Nousias[1], Adam Major[1],
Tughrul Arslan[1,2], Iain Lindsay[1]

[1]The University of Edinburgh
Mayfield Road, Edinburgh
United Kingdom, EH3 9JL
mark.muir@ed.ac.uk

[2]Institute for System Level Integration
Alba Centre, Livingston
United Kingdom, EH54 7EG

ABSTRACT

This paper describes a technique for automated dynamic structural-level pipelining of programs targeting dynamically reconfigurable processors with very short reconfiguration times. These architectures are particularly suited to streaming applications, whose primary market are low-cost, high-volume consumer products such as the image signal processor for digital cameras in modern mobile phones. These reconfigurable processors open up the ability for vendors to differentiate their products by providing their own algorithms. To minimise area (and thus cost), it is important that vendors have the ability to tailor the resources of the core to their needs. Therefore, the process of application development is that of hardware/software co-design. As part of a high-level software tool chain, we present an optimisation technique that can be added to the compiler to significantly improve the throughput of applications, by pipelining tight loops (kernels) which perform the majority of the work in streaming applications. This allows more complex algorithms to be deployed whilst still meeting the available timing budget. The timing constraint is determined automatically, in a manner which maximises throughput within a given resource budget.

1. INTRODUCTION

The choice of platform for many modern digital signal processing tasks in embedded systems is often limited to application-specific integrated circuits (ASICs), since off-the-shelf programmable architectures such as DSPs and microprocessors cannot meet the throughput requirements, whereas reconfigurable hardware such as field-programmable gate arrays (FPGAs) require too much area and power. However, for applications that demand an element of reprogrammability, streaming processors (such as those offered by Ambric [1] and SPI [2]) are becoming an increasingly attractive solution, which improve on throughput by providing multiple processing elements/cores with an interconnect structure suited to streaming. However, these processing elements—usually based on regular DSP designs—often equate to significant silicon area. Alternatively, coarse-grained dynamicaly reconfigurable architectures (DRAs) offer a high degree of parallelism, sufficient to achieve high throughput [3][4]. Thus fewer cores are required for a given application, leading to a much lower area overhead. These coarse-grained architectures, if given the ability to control their own reconfiguration, can be reconfigured very rapidly (e.g. millions of times per second), in order to achieve control flow similar to a regular microprocessor. This paper focuses on maximising the performance of programs running on a single core. However, the techniques can be directly applied to programs running on additional cores in a complete streaming application.

Coarse-grained DRAs, such as instruction cell based processors [5][6], provide a high degree of instruction chaining inside the core, by allowing arbitrary connections to be made between the various functional units via a configurable routing network. This allows quite complex data paths to be rendered onto the fabric and executed in a single configuration. This makes these architectures particularly suitable to stream processing, as fewer fetches from program memory are required. Performance is optimised by attempting to match the size of each kernel (inner loops where most of the execution time is spent) to the available resources, allowing them to fit into a single configuration context. This allows the configuration to persist for many clock cycles, operating on new data on each cycle. This increases throughput, since no time is spent having to reconfigure the core between successive iterations. It also decreases power consumption, as the configuration only needs to be fetched from program memory (or cache) once—upon first entering the kernel—rather than on every iteration. However, the resulting data paths can often have a long critical path, leading to poor temporal utilisation of the functional units, since they have to wait until all functional units have completed before operating on the next batch of data, which limits the throughput.

Pipelining provides a way of starting to operate on a new batch of data before an old one has completed. Thus, this allows the functional units of multiple stages of the kernel to be active concurrently; each operating on a different batch of data. Others have devised loop pipelining techniques for reconfigurable architectures [7, 8, 9], where successive iterations of the loop are replicated in hardware, and offset from each other to deal with any data dependencies between the iterations. These are most suitable for large reconfigurable architectures with much longer reconfiguration times, where there are sufficient resources for the entire loop body to be replicated many times. This paper elaborates on and extends work in a previous paper where structural-level pipelining techniques were shown to be applicable via software to rapidly reconfigurable/programmable architectures supporting operation-chaining. The technique allows complete kernels that were mapped to a single configuration context, to have their critical path length decreased by the addition of pipeline stage registers. Pipeline fill-

ing and flushing are achieved through dynamic reconfiguration.

The contribution in this work is the ability to automate the tasks of identifying configuration contexts which could benefit from pipelining, and choice of critical path constraint. In particular, in order to reduce power in the target architectures, the master clock frequency is kept as low as possible. Configuration contexts are allowed to persist for multiple clock cycles, until their critical path has completed. Pipelining reduces the critical path, so as a result, the quantisation introduced by the master clock frequency affects pipelined contexts more. Therefore, it is important to minimise the wasted time between the critical path stabilising and the next master clock cycle. The automatic pipelining algorithm demonstrated here attempts to do this.

Section 2 reviews existing pipelining techniques, and relevant software optimisation techniques. Section 3.1 describes an algorithm to perform pipeline stage allocation, and section 3.2 shows how properties of dynamic reconfiguration can be used to fill and flush the resulting pipeline. Section 3.3 details how the task can be completely automated. Section 4 shows the result of applying this technique to a real-life kernel used in image processing.

2. PREVIOUS WORK

For architectures that support instruction chaining, scheduling involves mapping as many dependent and independent data paths into as few configuration contexts as possible [10]. Independent data paths run in parallel, so the time for which a configuration persists is determined by the maximum critical path length of these data paths. If sufficient functional unit resources are available, loops can be optimised by loop unrolling [11]—i.e. placing multiple iterations as independent data paths in the same configuration. This allows multiple iterations to begin and end at once. This does not change the original critical path length, yet can increase the throughput. The throughput is determined by the critical path length of a loop iteration and the number of iterations that can be performed at once. During each execution of the loop configuration context, data propagates through the operation chains until the final result is ready. This means that the functional units involved in that chain are only performing useful work for a fraction of the time. This is where structural-level pipelining of these data paths comes in—to artificially reduce the critical path length by allowing new iterations to begin without waiting for the completion of previous iterations.

Various approaches of pipelining data paths have been proposed [12]. These require that the designer specifies a throughput constraint, in order to allow the algorithm to best make the choice between throughput and the area overhead each pipeline stage introduces. These approaches describe various algorithms for the task of pipeline stage allocation, applied to a number of different levels in a design. On reconfigurable architectures such as FPGAs, custom pipelines can be rendered as part of the configuration, leading to significant increases in throughput [13]. The previous work on dynamically pipelining DRAs [14] proposed a technique where pipelining would be performed based on a critical path constraint provided by the application developer. The work here elaborates on this technique, and looks into more detail on the real-life performance. Extensions are proposed to automate the choice of critical path constraint, and to maximise the real-life throughput.

3. DYNAMIC PIPELINING

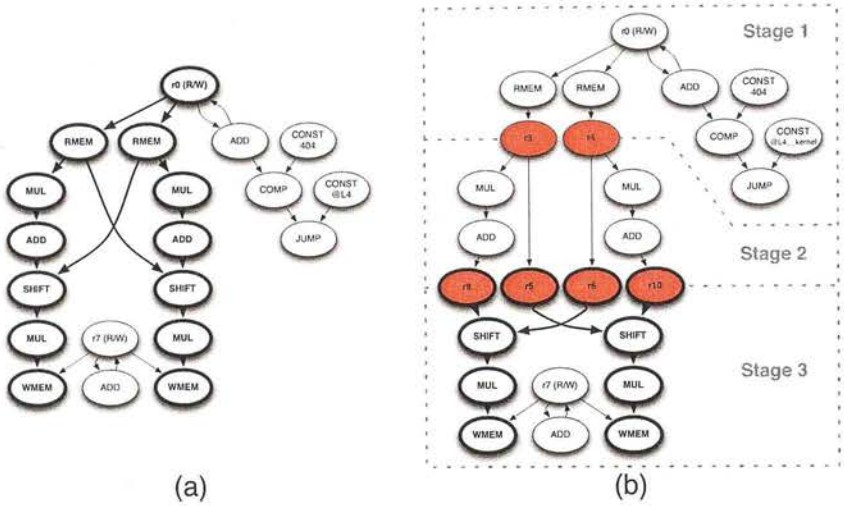
Conventional structural-level pipelining can be applied to single configuration context kernels with long critical data paths, in order to reduce the critical path, and thus increase throughput. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at runtime. This is done using existing register resources in the core to delay values for a single execution cycle, allowing values to be bridged across pipeline stage boundaries.

Structural pipelining is applied to the kernel basic block by first assigning each operation in the original data flow graph to a pipeline stage. Then, registers are introduced to store values over boundaries between pipeline stages. Only those values that are used in later pipeline stages are stored. A new register is needed for each value for each pipeline stage boundary over which it must persist. Figure 1 shows an example kernel before and after structural-level pipelining. The example includes only simple feedback chains consisting of a simple increment of the value of a register, however more complex feedback chains are also possible.

3.1 Pipeline stage allocation

First, constraints are defined between operations, where the order of execution is important. Examples include 'same stage or earlier' constraints between operations reading from input registers and operations that have those same registers marked as global output registers, and 'same stage or earlier' constraints between data memory read operations and potentially aliasing data memory write operations. All operations in a feedback chain must be placed in the same pipeline stage, since such chains require single-step total latency in order to keep the pipeline full. The algorithm for assigning pipeline stages to each operation is as follows:

- Identify the 'jump' operation, and all of its dependencies. Save this in a set—the 'jump chain' set.
- Create the 'remaining' set—a record of those operations yet to be assigned to a pipeline stage. This is initially populated with all the operations except for those in the 'jump chain' set.
- Define the constraints:
  - Add 'same stage or earlier' constraints between operations reading from input registers, and operations that have those same registers marked as global output registers.
  - Add 'same stage or earlier' constraints between data memory read operations and potentially aliasing data memory write operations.
  - Add 'same stage or earlier' constraints between volatile operations of the same kind, to ensure that they still appear in their original order.
- Detect feedback chains:
  - Identify all the operations that are part of each feedback chain, and record them in a set for each chain. These shall be referred to as the 'feedback' sets. No operation in a feedback set may be assigned to a pipeline stage until all the operations in that set are ready to be assigned.
- Create an ordered list of pipeline stages, initially consisting of a single entry. Each entry contains the set of operations that have been assigned to that pipeline stage.
- For each operation in the 'remaining' set:
  - Create a temporary set containing this operation and any operations in the same 'feedback' set (if one exists).
  - Determine whether any of the operations in the temporary set have any successors that are also in the 'remaining' set. If they do, then the temporary set is not ready, so discard it and move on to the next operation in the 'remaining' set.



1: Example kernel data flow graph, (a) before pipelining, (b) after pipelining (kernel loop context). The inserted pipeline stage registers are shown in red. The per-cycle critical path is shown in bold, and is shorter in (b), which allows for a higher throughput.

- Determine whether any constraints involving the operations in the temporary set involve operations that are also in the 'remaining' set. If they do, then the temporary set is not ready, so discard it and move on to the next operation in the 'remaining' set.
- Identify the latest pipeline stage where all the operations in the temporary set could be placed, according to their dependencies and constraints.
- Construct a configuration context containing all the pipeline stages constructed thus far, and calculate its critical path delay (including the reading from and writing to pipeline registers).
- Speculatively construct a configuration context containing all the pipeline stages constructed thus far, including the operations from the temporary set, placed in the previously identified pipeline stage. Calculate its critical path delay.
- If the critical path delay is different (i.e. increased), and the new delay exceeds the target, then move to the preceding pipeline stage (creating a new pipeline stage at the beginning of the list, if the chosen stage was the first in the list).
- Transfer the operations from the temporary set to the identified pipeline stage, and remove them from the 'remaining' set.
- Loop whilst the 'remaining' set is not empty.

- Add the operations from the 'jump chain' set to the first pipeline stage.

The algorithm is a form of list scheduling. Only operations whose predecessors (in the data path) have already been assigned a pipeline stage may be considered for insertion on each pass. In order to minimise the register count, operations should be placed in as late a pipeline stage as possible. Operations that must be placed in the same stage are dealt with together. Operations are considered for placement in the latest pipeline stage containing any of their predecessors. Then, the insertion point is moved towards later pipeline stages until all constraints have been satisfied. Once a valid insertion point has been identified, the critical path is calculated for the resulting (incomplete) configuration context with the operation in that pipeline stage. If the critical path meets the target value, the operation is placed in that pipeline stage. Otherwise, the operation is added to the next pipeline stage (creating it if it does not exist).

The creation of dependencies ensures that the sequence of state changes is maintained, thus ensuring correct results. Assigning operations to a late a pipeline stage as possible aids to reduce the number of registers required. Once the pipeline stages have been determined, pipeline stage registers are assigned as follows:

- For each pipeline stage in sequence:
  - Assign a new register storing the value produced by each operation in all previous pipeline stages that needs to be stored for use in this or any later stage.

3.2 Dynamic initialisation and clean-up

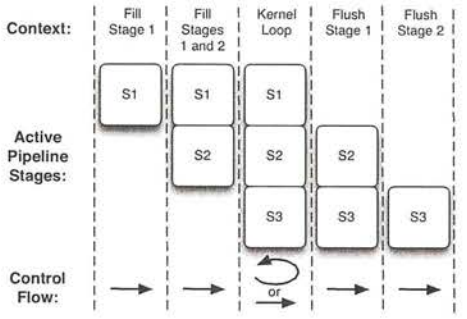
Normally, a pipelined design would require additional logic to take care of initialising the pipeline stages, or to suppress the operations in later pipeline stages until the previous stages have filled (predication), so that they do not operate on garbage. However, the pipelines in a coarse-grained DRA are themselves rendered as part of the configuration context. Provided that the configuration time is not significantly larger than the execution time of each step, dynamic reconfiguration can be used to render different configurations before the main kernel loop configuration, to fill successive stages of the pipeline, and similarly to flush the pipeline after exiting the kernel loop. This allows the kernel loop configuration to assume that the pipeline stages are always full. This provides a generic, purely software alternative to predication, which can be used as a fall-back when no hardware support exists.

Prologue: New configuration contexts are created to initially fill each successive stage of the pipeline. For n pipeline stages, $n-1$ pipeline filling contexts are created.

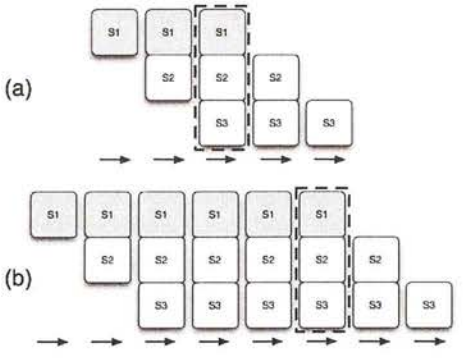
Loop: A single configuration context is created for the kernel loop, which includes all pipeline stages.

Epilogue: New configuration contexts are created to flush successive stages of the pipeline. For n pipeline stages, $n-1$ pipeline flushing contexts are created.

The core is dynamically reconfigured to first perform pipeline initialisation, then reconfigured to execute the kernel loop, then finally reconfigured to flush the pipeline—as demonstrated in figure 2. This is similar to the epilogue and prologue in software pipelining [15].



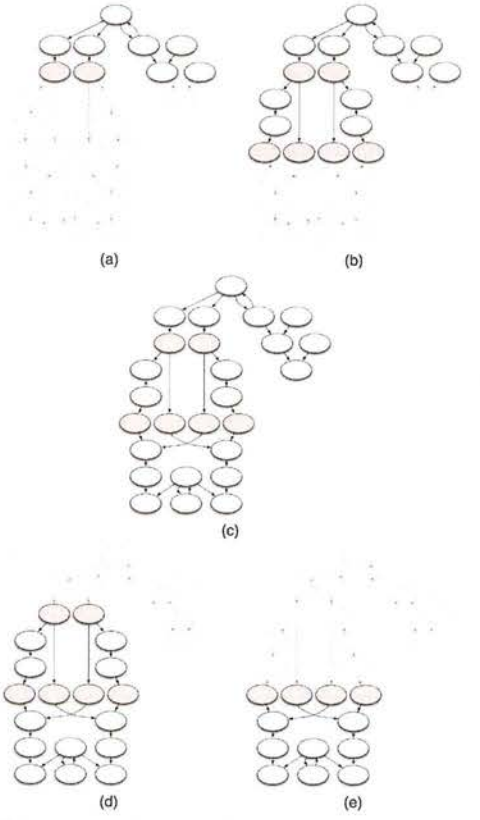
2: Control flow for a 3-stage pipelined kernel, showing which stages are active in each context (and moment in time). Execution flows from one context to the next, except in the kernel loop, which loops back to itself (holding the same context) until the end condition is satisfied.



3: Expanded control flow for the pipeline shown in figure 2 for (a) 3, and (b) 6 iterations. The point at which the loop termination condition should evaluate to true is shown by a dotted box. It can be seen in both cases that only the first stage has executed for the desired number of iterations by this point.

The configuration contexts generated for the kernel example from figure 1 is shown in figure 4. The use of separate special-purpose configurations alleviates the need for special logic for this purpose in the kernel loop configuration context, keeping its size down, and thus not compromising the potential parallelism in the core.

Figure 2 shows which stages of the pipeline are active during execution for a 3-stage pipeline. As the target architectures may not be state free (e.g. memory access), it is important to not allow any operation in any pipeline stage to operate on garbage, and to preserve the execution count. With the arrangement shown in the figure, all pipeline stages will be executed the same number of times irrespective of the number of iterations performed in the kernel loop.



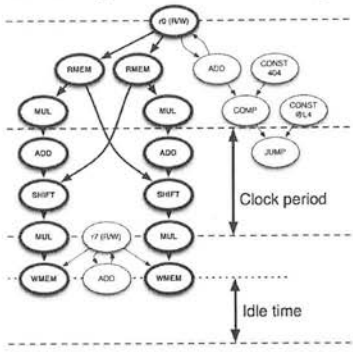
4: The sequence of configuration contexts created for the example kernel, (a) iteration 1—filling pipeline stage 1, (b) iteration 2—filling pipeline stages 1 and 2, (c) iterations 3 to $n-2$—pipeline full (loop), (d) iteration $n-1$—flushing pipeline stage 1, (e) iteration n—flushing pipeline stage 2.

Now consider the original kernel, where the 'jump' operation causes the loop to terminate after n iterations. In the pipelined kernel, we must ensure that the kernel loop terminates after n executions of the operations that calculate the loop termination condition; otherwise, the operations or operands would need to be modified to yield a different iteration count. Looking at figure 2, the minimum number of iterations possible in the pipelined design occurs when the kernel loop context executes only once. This corresponds to an iteration count equal to the number of pipeline stages (in this case 3). In order for the loop to terminate immediately, the operations that determine the loop termination condition must have been executed this number of times by the time the kernel loop context has been executed. This can only be achieved by placing these operations in the *first* pipeline stage. The same argument also applies for any higher iteration count. Figure 3 shows two examples, to highlight this point.

Placing the 'jump' in the first pipeline stage therefore requires that all of its dependencies are also placed in the first pipeline stage. Since the pipeline filling contexts (prologue) should always be executed in sequence (with no branching), the 'jump' operation is omitted from these contexts, even though it is in a pipeline stage

active in those contexts. Its dependencies are left in place, since their side effects are important—e.g. they could update the iteration counter whose value is used to determine the loop termination condition.

3.3 Automating the choice of timing constraint



5: Idle time resulting from the master clock. The shorter the critical path of the kernel, the more effect this has. This particularly affects pipelined kernels.

The arbitrary operation chaining supported by the target architectures leads to a great variation in critical path length in different configuration contexts, as paths can be constructed involving long chains of a varying number of cells, and each type of cell has a different combinatorial delay. Ideally, each iteration of the configuration context should be allowed to persist for the time required for the results to stabilise on the operation(s) that lie at the end of the critical path. In order to avoid the overhead of asynchronous logic, a master clock is normally used instead, and the iteration ends on the next master clock cycle after the last results have stabilised, as can be seen in figure 5. To minimise the resulting idle time between these two events, it is desirable to minimise the period of the master clock. However, high clock frequencies come at the cost of power consumption and area. Therefore, a suitable compromise has to be made.

Since pipelining reduces the critical path length of each iteration of the kernel loop configuration context, the quantisation introduced by the master clock frequency affects pipelined contexts more. Therefore, it is important to minimise the wasted time between the critical path stabilising and the next master clock cycle. This fact is used to aid the automatic choice of the timing constraint.

The timing constraint is initially chosen to be the minimum possible critical path length that a pipeline stage can consist of. This is determined by the length of certain data paths that cannot be split across pipeline stages. These include the jump condition logic determining when to finish the loop, and feedback loops that update a register or memory location (where that register or memory location is both read from and written to in the same kernel). The one with the longest critical path length is selected, and the value rounded up to the next integer multiple of the master clock period.

Then, pipeline stage allocation is performed using this critical path constraint. If a valid pipeline could be constructed, register allocation is performed. If there are sufficient registers available, then this pipeline geometry is used, since it will result in the highest possible iteration rate. Otherwise, the timing constraint is incremented by one master clock period, and the process continues. A natural end point exists where this value reaches the critical path of the non-pipelined kernel. If reached, the context is left non-pipelined.

For completely automatic pipelining, feedback-directed optimisation is used. The program is first executed in a simulator prior to pipelining, and profiling information is fed back into the compiler. Basic blocks that loop to themselves are identified, and where sufficient resources exist in the core to map the entire block into one configuration context, these are potential candidates for pipelining. The number of consecutive iterations of each candidate is determined through the profiling results. The minimum consecutive iterations for a kernel defines the maximum depth to which it can be pipelined: the pipeline depth must not be less than the minimum execution count. This is used as a test during each iteration of the timing constraint selection algorithm, where a potential pipeline is checked for its depth not exceeding the minimum iteration count. If it does, then the geometry is considered invalid, and the algorithm continues with a larger timing constraint. To take into account the cost of loading the new configurations from memory, the minimum iteration count value is artificially reduced by an arbitrary count, to weigh the algorithm in favour of only pipelining loops with significant iteration counts.

4. APPLICATION TO STREAMING

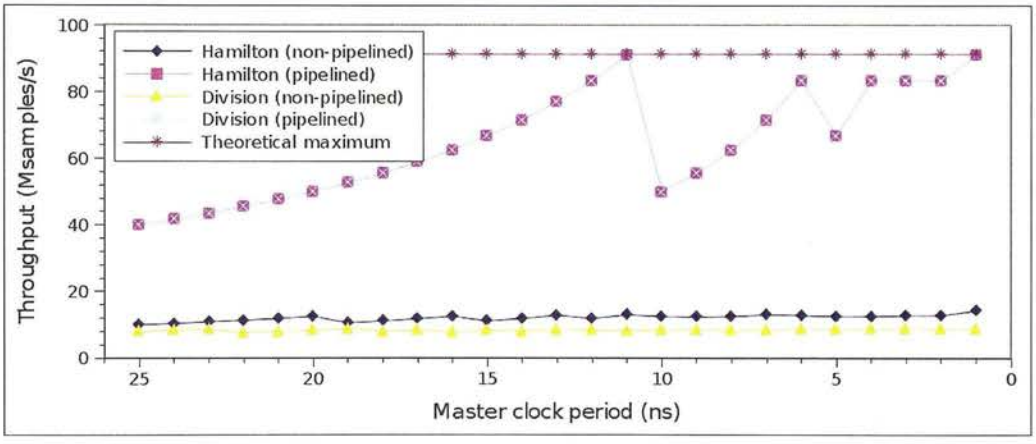
The algorithm described in this paper was applied to two real-life applications: a 7-line Hamilton demosaic filter [16], and a multiplication-based iterative software division algorithm. The demosaic involves interpolating missing colour components from the Bayer output of a colour filter array sensor. Division on a per-pixel level is used as part of many commercial noise reduction filters. Both are high-throughput tasks normally done on-chip as part of a custom image signal processing (ISP) pipeline, used in modern digital cameras and mobile phones. Both kernels were implemented on a reconfigurable instruction cell-based processor [5] (180nm timing figures), using the C language. Software optimisation techniques were used to reduce the main kernel in each case into a basic block small enough to fit onto the target architecture in a single configuration context. Both example kernels produce a single output pixel per iteration.

The performance of the pipelining for both cases is shown in figure 6, and some additional details are given for the Hamilton demosaic in table 1.

The main trend to notice is the ability for the maximum achievable iteration rate (after pipelining) to generally increase as the master clock frequency is increased. Since the same underlying data path is used in each case, the non-pipelined critical path length is constant. The iteration time of the non-pipelined data paths is just the critical path length rounded up to the next integer multiple of the master clock period. As the master clock period is decreased, the algorithm is able to produce a pipeline with a critical path closer to the theoretical minimum (as dictated by the indivisible data paths such as feedback loops, and the jump condition chain). However, the number of pipeline stages required to do this increases in a faster than linear fashion. This is due to quantisation: the error between the time taken for each data path fragment in each pipeline stage to complete and the closest integer multiple of the master clock frequency. As the pipeline stages get shorter, the relative size of the indivisible units being pipelined (i.e. the internal delays of each cell and section of interconnect) increases compared to the resolution of the master clock. The algorithm does well in minimis-

Master clock period (ns)	20.0	15.0	10.0	5.0	3.0	2.0	1.0
Pipeline stages	5	7	5	7	9	9	11
Pipeline stage registers	80	123	80	123	153	153	189
Min. possible constraint (ns)	10.95	10.95	10.95	10.95	10.95	10.95	10.95
Non-pipelined critical path (ns)	77.0	77.0	77.0	77.0	77.0	77.0	77.0
Pipelined critical path (ns)	19.8	14.65	19.8	14.65	11.55	11.55	11.00
Improvement in critical path	389%	526%	389%	526%	667%	667%	700%
Non-pipelined iteration time (ns)	80.0	90.0	80.0	80.0	78.0	78.0	77.0
Pipelined iteration time (ns)	20.0	15.0	20.0	15.0	12.0	12.0	11.0
Improvement in iteration time	400%	600%	400%	533%	650%	650%	636%
Pipelined throughput (MPixels/s)	50.0	66.6	50.0	66.6	83.3	83.3	90.9

1: Performance of the demosaic filter kernel before and after automatic pipelining, over a range of master clock periods. See section 4 for an explanation of the results.



6: Throughput before and after automatic pipelining, over a range of master clock periods, for two pixel-level code examples: Hamilton demosaic and iterative software division. The theoretical line shows what could be achieved if the master clock were of infinite frequency, based on the longest indivisible critical path (the iteration control logic in both of these cases).

ing this effect, and the percentage improvements with and without the effect of the master clock are relatively close in all cases.

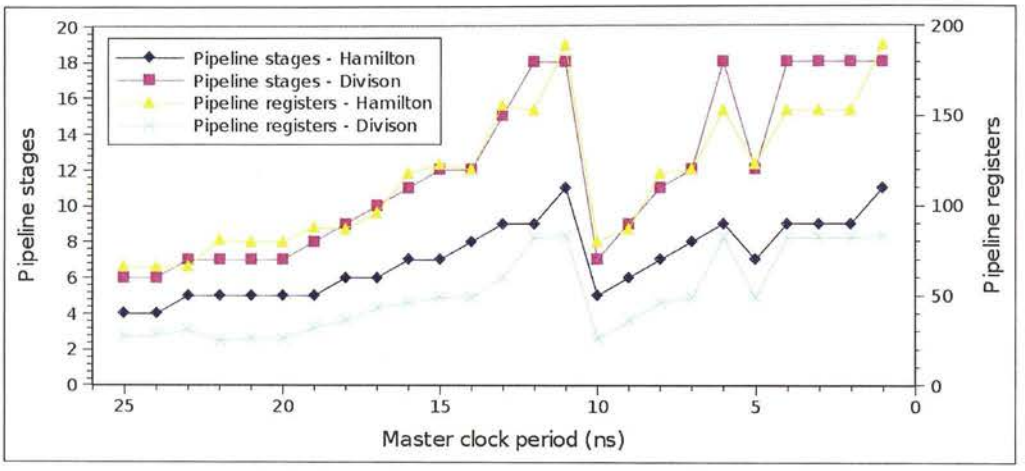
The pipeline geometries contructed for each master clock frequency setting are shown in figure 7. Both examples show identical post-pipelining throughput (iteration rate), as both cases have the same longest indivisible critical path—corresponding to the iteration control (jump) logic (shown by the theoretical line in figure 6). There are no data dependencies or other constraints limiting the potential for pipelining in either example. If data dependencies, feedback loops, or other constraints were present, these would be reflected by a larger indivisible critical path. The shorter the indivisible critical path, the more important the behaviour of the automatic pipelining algorithm.

The resource-saving effect of the algorithm can be seen to come into effect each time the current integer multiple of the master clock frequency drops below the indivisible critical path length. This makes the iteration rate curve appear to wrap around each time it tries to cross the theoretical maximum iteration rate line. By extending the length of the pipeline stages up to the next master clock period, the number of registers is minimised, which avoids needless congestion on the interconnect. The reduction in the number of pipeline stages reduces the configuration size and the latency, since fewer filling and flushing iterations need to be performed.

5. CONCLUSIONS

This work proposed an algorithm for automatically applying dynamic structural-level pipelining to single configuration context kernels running on dynamically reconfigurable arrays (DRAs). The technique is a form of feedback directed optimisation, where profiling information (consecutive execution counts) are used to determine which kernels will benefit from pipelining. Candidates with very low consecutive execution counts must not be pipelined too deeply. This is to ensure that the additional latency of pipeline filling and flushing is more than nullified by the decrease in total execution time for the pipelined kernel loop when the pipeline is full. This is only possible when the minimum possible iteration count is known. This is the case for pixel-level kernels in the ISP application domain, as the iteration count is typically the line size of the image.

An iterative approach is used to form an efficient pipeline, where the timing constraint is automatically chosen to be an integer multiple of the master clock frequency. The timing constraint is incre-



7: Pipeline geometry from automatic pipelining, over a range of master clock periods, for two pixel-level code examples: Hamilton demosaic and iterative software division.

mented until a valid pipeline can be constructed without encountering register starvation. The range of possible pipeline geometries is controlled by the availability of registers. Architectures with distributed registers will offer the best results, otherwise the bandwidth of the interface and/or additional combinatorial delays introduced by routing to and from a register file would likely outweigh any benefit. This makes the case for registers to be made available in the interconnect itself.

The algorithm was applied to a demosaic kernel of modest complexity and to a software division algorithm, leading to the possibility to pipeline to a significant depth. A performance increase of up to 7 times can be obtained for the demosaic example, and nearly 10 times for the division. As the pipeline gets deeper, the cost—in terms of register requirement and storage for pipeline filling and flushing contexts—increases more than linearly. As the critical path of the pipelined kernel gets smaller, the quantisation of the iteration rate caused by the master clock, gets increasingly worse. Inside the bounds of this quantisation, reducing the pipeline critical path (by increasing the number of pipeline stages) has no effect on the iteration rate. In these situations, extra resources would be introduced for no benefit. To avoid this, the proposed algorithm relaxes the critical path to take into account this quantisation, thus minimising the resource requirements for a given physically achievable iteration rate.

6. REFERENCES

[1] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, 2007, pp. 55–64.

[2] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Daly, "A programmable 512 GOPS stream processor for signal, image, and video processing," in *Solid-State Circuits Conference*, 2007, pp. 272–602.

[3] A. Major, T. Arlsan, *et al.*, "H.264 decoder implementation on a dynamically reconfigurable instruction cell based architecture," in *International SOC Conference*, 2006, pp. 49–52.

[4] Z. Khan, T. Arlsan, *et al.*, "Implementation of a real time programmable encoder for low density parity check code on a reconfigurable instruction cell architecture," in *Design Automation Conference, Asia and South Pacific*, 2007, pp. 583–588.

[5] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 1–11, 2008.

[6] "Loosely-biased heterogeneous reconfigurable arrays," U.S. Patent 20 050 257 024, 2005.

[7] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Trans. Comp.-Aid. Des. Integ. Circ. and Syst.*, vol. 20, no. 2, pp. 234–248, 2001.

[8] J. Liao, W. Wong, and T. Mitra, "A model for hardware realization of kernel loops," in *Field Programmable Logic, International Conference on*, 2003, pp. 334–344.

[9] R. Rodrigues and J. Cardoso, "Pipelining sequences of loops—a first example," in *ARC, Workshop*, 2005, pp. 147–151.

[10] Y. Yi and I. Nousias, "System-level scheduling on instruction cell based reconfigurable systems," in *Design Automation and Test in Europe, International Conference on*, 2006, pp. 381–386.

[11] J. Sanchez and A. Gonzalez, "The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures," in *ICCP Parallel Processing, International Conference on*, 2000, p. 555.

[12] S. Bakshi and D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 4, pp. 419–432, 1999.

[13] S. Silva and S. Bampi, "Area and throughput trade-offs in the design of pipelined discrete wavelet transform architectures," in *Design Automation and Test in Europe, International Conference on*, 2005, pp. 32–37.

[14] M. Muir, T. Arslan, and I. Lindsay, "Automated dynamic throughput-constrained structural-level pipelining in streaming applications," in *Design Automation and Test in Europe, international conference on*, 2008, pp. 1358–1361.

[15] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *ACM SIGPLAN conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.

[16] R. Ramanath, W. Snyder, and G. Bilbro, "Demosaicking methods for bayer color arrays," *Electronic Imaging*, vol. 11, no. 3, pp. 306–315, 2002.

References

[1] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, 2007, pp. 55–64.

[2] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Daly, "A programmable 512 GOPS stream processor for signal, image, and video processing," in *Solid-State Circuits Conference*, 2007, pp. 272–602.

[3] A. Major, T. Arlsan, *et al.*, "H.264 decoder implementation on a dynamically reconfigurable instruction cell based architecture," in *International SOC Conference*, 2006, pp. 49–52.

[4] Z. Khan, T. Arlsan, *et al.*, "Implementation of a real time programmable encoder for low density parity check code on a reconfigurable instruction cell architecture," in *Design Automation Conference, Asia and South Pacific*, 2007, pp. 583–588.

[5] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 1–11, 2008.

[6] "Loosely-biased heterogeneous reconfigurable arrays," U.S. Patent 20 050 257 024, 2005.

[7] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Trans. Comp.-Aid. Des. Integ. Circ. and Syst.*, vol. 20, no. 2, pp. 234–248, 2001.

[8] J. Liao, W. Wong, and T. Mitra, "A model for hardware realization of kernel loops," in *Field Programmable Logic, International Conference on*, 2003, pp. 334–344.

[9] R. Rodrigues and J. Cardoso, "Pipelining sequences of loops—a first example," in *ARC, Workshop*, 2005, pp. 147–151.

[10] M. Muir, T. Arslan, and I. Lindsay, "Automated dynamic throughput-constrained structural-level pipelining in streaming applications," in *Design Automation and Test in Europe, international conference on*, 2008, pp. 1358–1361.

[11] M. Muir, N. Aslam, I. Nousias, A. Major, T. Arslan, and I. Lindsay, "Automatic dynamic structural-level pipelining in reconfigurable processors," in *Design and Architectures for Signal and Image Processing, conference on*, 2008, pp. 222–228.

[12] M. Muir, I. Lindsay, T. Arslan, I. Nousias, S. Khawam, M. Milward, N. Aslam, and A. Major, "Extensible software emulator for reconfigurable instruction cell based processors," in *SOC conference, IEEE international conference on*, 2008, pp. 35–40.

[13] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM, 1999, pp. 610–615.

[14] J. Henkel, "Closing the SoC design gap," *IEEE computer*, 2003.

[15] K.-C. Wu and Y.-W. Tsai, "Structured ASIC, evolution or revolution?" in *ISPD '04: Proceedings of the 2004 international symposium on Physical design*. New York, NY, USA: ACM, 2004, pp. 103–106.

[16] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, Bourgeault, *et al.*, "The Stratix II logic and routing architecture," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2005, pp. 14–20.

[17] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in Virtex-II FPGA family," in *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2002, pp. 157–164.

[18] A. Gayasen, N. Vijaykrishnan, and M. Irwin, "Exploring technology alternatives for nano-scale FPGA interconnects," June 2005, pp. 921–926.

[19] "1 GHz field programmable object array overview," MathStar, 2007.

[20] P. Chiang and S. Riley, "Using a field programmable object array (FPOA) to accelerate image processing," in *Real-Time image processing*, vol. 6063, no. 1. SPIE, 2006, p. 60630E.

[21] K. Underwood and K. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance," April 2004, pp. 219–228.

[22] M. Herbordt, T. VanCourt, Y. Gu, and B. Sukhwani, "Achieving high performance with FPGA-based computing," *Computer: IEEE computiing society*, 2007.

[23] J. Rice, K. Pace, M. Gates, G. Morris, and K. Abed, "Reconfigurable computer application design considerations," April 2008, pp. 236–243.

[24] P. Martin, M. Smith, S. Alam, and P. Agarwal, "Implementation methodology for emerging reconfigurable systems," Aug. 2008, pp. 169–172.

[25] P. Claydon, "Multicore future is right now," in *EE Times-Asia*, 2007.

[26] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, 1995.

[27] J. Stark, M. Evers, and Y. N. Patt, "Variable length path branch prediction," *SIGPLAN Not.*, vol. 33, no. 11, pp. 170–179, 1998.

[28] J. Farrell and T. Fischer, "Issue logic for a 600-MHz out-of-order execution microprocessor," *Solid-State Circuits, IEEE Journal of*, vol. 33, no. 5, pp. 707–712, May 1998.

[29] M. Lipasti and J. Shen, "Superspeculative microarchitecture for beyond AD 2000," *Computer*, vol. 30, no. 9, pp. 59–66, Sep 1997.

[30] P. Hoare, A. Jones, D. Kusic, *et al.*, "Rapid VLIW processor customization for signal processing applications using combinational hardware functions," *EURASIP Journal on Applied Signal Processing*, vol. 2006, 2006.

[31] T. Halfhill, "Silicon Hive breaks out," *Microprocessor Report*, no. 169, December 2003.

[32] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *ACM SIGPLAN conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.

[33] D. W. Wall, "Limits of instruction-level parallelism," in *The Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 176–188.

[34] W. A. Wolf and S. A. McKee, "Hitting the memory wall: implications of the obvious," in *SIGRAPH Comput. Archit. News*, vol. 23, 1995, pp. 20–24.

[35] E. Kilgariff and R. Fernando, "The GeForce 6 series GPU architecture," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM, 2005, p. 29.

[36] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM, 2003, pp. 917–924.

[37] J. Fung and S. Mann, "Using multiple graphics cards as a general purpose parallel computer: applications to computer vision," vol. 1, Aug. 2004, pp. 805–808.

[38] A. Duller, D. Towner, and G. Panesar, "picoArray technology: the tool's story," in *Design, Automation and Test in Europe*, vol. 3, 2005, pp. 1530–1591.

[39] "Coupling integrated circuits in a parallel processing environment," U.S. Patent 7 539 845, 2006.

[40] X. Jia and R. Vemuri, "Using GALS architecture to reduce the impact of long wire delay on FPGA performance," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2005, pp. 1260–1263.

[41] "Xtensa processor," Tensilica Inc. [Online]. Available: http://www.tensilica.com

[42] "ARCtangent processor," ARC Intl. [Online]. Available: http://www.arc.com

[43] "Stretch processor," Stretch Inc. [Online]. Available: http://ww.stretchinc.com

[44] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," *Field-Programmable Logic and Applications*, pp. 61–70, 2003.

[45] L. Bauer, M. Shafique, and J. Henkel, "Run-time instruction set selection in a transmutable embedded processor," in *DAC '08: Proceedings of the 45th annual conference on Design automation*. New York, NY, USA: ACM, 2008, pp. 56–61.

[46] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 642–649.

[47] P. Heysters, G. Smit, and E. Molenkamp, "A flexible and energy-efficient coarse-grained reconfigurable architecture for mobile systems," *The Journal of Supercomputing*, vol. 26, no. 3, pp. 283–308, 2003.

[48] H. Schmitt, D. Whelihan, *et al.*, "PipeRench: A virtualised programmable datapath in 0.18 micron technology," in *Custom Integrated Circuits Conference*, 2002, pp. 63–66.

[49] S. Khawam, "Reconfigurable architectures for low-power SoC: Domain- specific and rica based systems," Ph.D. dissertation, University of Edinburgh, School of Engineering, apr 2006.

[50] P. Bellows and B. Hutchings, "JHDL - an HDL for reconfigurable systems," in *FPGAs for Custom Computing Machines, IEEE symposium on*, 1998, pp. 175–184.

[51] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," April 2006, pp. 45–56.

[52] A. Takach, B. Bowyer, and T. Bollaert, "C based hardware design for wireless applications," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 124–129.

[53] K. Hammond and G. Michaelson, "Bounded space programming using finite state machines and recursive functions: the hume approach," in *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2006.

[54] A. Hoffman, T. Kogel, and H. Meyr, "A framework for fast hardware-software co-simulation," in *Design Automation and Test in Europe, international conference on*, 2001, pp. 760–764.

[55] A. Halambi, P. Grun, *et al.*, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Design Automation and Test in Europe, international conference on*, 1999, pp. 485–490.

[56] E. R. Altman, S. Sathaye, and M. Gschwind, "Execution-based scheduling for VLIW architectures," in *Euro-Par'99—Parallel Processing, international conference on*, 1999, pp. 1269–1275.

[57] Y. Yi and I. Nousias, "System-level scheduling on instruction cell based reconfigurable systems," in *Design Automation and Test in Europe, international conference on*, 2006, pp. 381–386.

[58] D. Novillo, "TreeSSA - a new high-level optimisation framework for the GNU compiler collection," 2003.

[59] ——, "Design and implementation of TreeSSA," 2004.

[60] C. Lattner and V. Adve, "The LLVM compiler framework and infrastructure tutorial," in *Mini Workshop on Compiler Research Infrastructures (LCPC'04)*, 2004.

[61] I. Nousias, "Reconfigurable computing: The reconfigurable instruction cell array: Reconfiguration and interconnects," Ph.D. dissertation, University of Edinburgh, School of Engineering, apr 2009.

[62] L. Robertson, "Anecdotes," *Annals of the History of Computing, IEEE*, vol. 27, no. 2, pp. 82–84, 2005.

[63] H. Diab and I. Demashkieh, "A reconfigurable microprocessor teaching tool," in *Science, Measurement and Technology, IEE Proceedings*, 1990, pp. 287–292.

[64] C. Cooper and P. Werstein, "The use of Java to develop a microprocessor emulator," in *Software Engineering: Education and Practice*, 1998, pp. 272–277.

[65] W. Zaatar and G. E. Nasr, "An implementation scheme for a microprocessor emulator," in *ICECS Electronics, Circuits and Systems, 7th international conference on*, 2000, pp. 169–172.

[66] S. Bush, "ARM offers real-time prototyping capability," *Electronics Weekly*, no. 39339, July 2006.

[67] R. Cohn and P. G. Lowney, "Feedback directed optimisation in compaq's compilation tools for Alpha," in *Proceedings of the 2nd ACM Workshop on Feedback-directed optimisation*, 1999.

[68] M. Gschwind, V. Salapura, and D. Maurer, "FPGA prototyping of a RISC processor core for embedded applications," *IEEE Transactions on VLSI Systems*, pp. 241–250, 2001.

[69] Y. Nakamura and K. Hosokawa, "Fast FPGA emulation based simulation environment for custom processors," *IEICE transactions on fundamentals of electronic communications in computer science*, vol. E89-A, pp. 3464–3470, 2006.

[70] S. Fink and E. Sanchez, "Development and prototyping for an 8-bit multitask micropower processor," in *Proceedings of the 6th IEEE International Workshop on Rapid System Prototyping*, 1995, pp. 75–78.

[71] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[72] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[73] The GNU compiler collection (open source). The free software foundation. [Online]. Available: http://gcc.gnu.org

[74] H. Nilsson, "Porting GCC for dunces," Axis Communication, 2000. [Online]. Available: http://ftp.axis.com/pub/users/hp/pgccfd/pgccfd-0.5.pdf

[75] M. Ganguin, M. Schinz, P. Mudry, and A. Ijspeert, "GCC back-end for the Ulysse processor," 2007. [Online]. Available: http://birg.epfl.ch/webdav/site/birg/users/146738/public/movegcc.pdf

[76] L. V. Put, D. Chanet, B. D. Bus, B. D. Sutler, and K. D. Bosschere, "DIABLO: a reliable, retargetable and extensible link-time rewriting framework," in *Proceedings of the 5th International Symposium on Signal Processing and Information Technology*, 2005, pp. 7–12.

[77] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen, "Directed hypergraphs and applications," *Discrete Applied Mathematics*, vol. 42, no. 2-3, pp. 177–201, 1993.

[78] J. A. DeRosa and H. M. Levy, "An evaluation of branch architectures," in *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1987, pp. 10–16.

[79] V. Bala and N. Rubin, "Efficient instruction scheduling using finite state automata," *International Journal of Parallel Programming*, vol. 25, no. 2, 1997.

[80] P. Duhamel and C. Guillemot, "Polynomial transform computation of the 2-D DCT," in *Acoustics, Speech and Signal Processing (ICASSP), international conference on*, vol. 3, 1990, pp. 1515–1518.

[81] W. Kao, S. Wang, L. Chen, and S. Lin, "Design considerations of color image processing pipeline for digital cameras," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 4, pp. 1144–1152, nov 2006.

[82] A. Chihoub, Y. Bai, and V. Ramesh, "An imaging library for a TriCore based digital camera," in *Proceedings of the 5th IEEE international workshop on computer architectures for machine perception (CAMP)*, 2000, pp. 3–11.

[83] D. Coffin. dcraw project (open source). [Online]. Available: http://www.cybercom.net/~dcoffin/dcraw/

[84] G. C. Fox, "What have we learnt from using real parallel machines to solve real problems?" in *Proceedings of the third conference on Hypercube concurrent computers and applications*. New York, NY, USA: ACM, 1988, pp. 897–955.

[85] S. Note, F. Catthoor, *et al.*, "Combined hardware selection and pipelining in high-performance data-path design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, no. 4, pp. 413–423, 1992.

[86] S. Bakshi and D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 4, pp. 419–432, 1999.

[87] S. Bakshi and D. D. Gajski, "Performance-constrained hierarchical pipelining for behaviors, loops, and operations," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 1, pp. 1–25, 2001.

[88] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[89] S. Bakshi, D. Gajski, and H. Juan, "Component selection in resource shared and pipelined DSP applications," in *EURO-DAC/EURO-VHDL: Proceedings of the conference on European design automation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996, pp. 370–375.

[90] S. Silva and S. Bampi, "Area and throughput trade-offs in the design of pipelined discrete wavelet transform architectures," in *Design Automation and Test in Europe, International Conference on*, 2005, pp. 32–37.

[91] C. Soviani, I. Hadzic, and S. Edwards, "Synthesis of high-performance packet processing pipelines," in *Design Automation and Test in Europe, International Conference on*, 2006, pp. 679–682.

[92] J. Sanchez and A. Gonzalez, "The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures," in *ICCP Parallel Processing, International Conference on*, 2000, p. 555.

[93] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," *SIGARCH Comput. Archit. News*, vol. 4, no. 4, pp. 159–164, 1976.

[94] B. R. Rau and J. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 9–50, 1993.

[95] F. Warg and P. Stenstrom, "Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms," in *Parallel Architectures and Compilation Techniques, Conference on*, 2001, pp. 221–230.

[96] "Execution unit chaining for single cycle extract instruction having one serial shift left and one serial shift right execution units," U.S. Patent 6 061 780, 2000.

[97] J. Mukherjee, M. Moore, and S. Mitra, "Color demosaicing with constrained buffering," in *Signal Processing and its Applications, Sixth International Symposium on*, vol. 1, 2001, pp. 52–55.

[98] R. Ramanath, W. Snyder, and G. Bilbro, "Demosaicking methods for bayer color arrays," *Electronic Imaging*, vol. 11, no. 3, pp. 306–315, 2002.

[99] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, "Fast and accurate simulation using the LLVM compiler framework," in *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO'09*, 2009.

[100] J. eun Lee, K. Choi, and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *Design and Test of Computers, IEEE*, vol. 20, no. 1, pp. 26–33, Jan-Feb 2003.

[101] J. Merrill, "GENERIC and GIMPLE: A new tree representation for entire functions," 2003.

[102] S. Callanan, D. Dean, and E. Zadok, "Extending GCC with modular GIMPLE optimisations," 2003.

[103] M. Sánchez-Élez, M. Férnandez, R. Maestre, F. Kurdahi, R. Hermida, and N. Bagherzadeh, "A complete data scheduler for multi-context reconfigurable architectures," in *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2002, p. 547.

[104] S. Lee, D. Raila, and V. Kindratenko, "LLVM-CHiMPS - compilation environment for FPGAs using LLVM compiler infrastructure and CHiMPS," in *Proceedings of the 4th annual reconfigurable systems summer institute RSSI'08*, 2008.